ELSEVIER

Science of Computer Programming 34 (1999) 79–140

Science of Computer Programming

www.elsevier.nl/locate/scico

A refinement calculus for tuple spaces

Laura Semini *, Carlo Montangero

Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56100 Pisa, Italy

Communicated by K. Apt; received 15 March 1997; received in revised form 26 August 1998

Abstract

It is fairly accepted that the realization of complex systems must be accomplished step by step from the initial specification, through a sequence of intermediate phases, to the final program. These development steps, linking a preliminary version, or description, of the program to a more detailed one, are usually called *refinement* steps, while the intermediate stages of a refinement process are called *levels of abstraction*.

A *refinement calculus* is a means to support this modus operandi in program development, allowing linking different levels of abstraction: it introduces a precise relation between intermediate descriptions, and the rules to check whether the relation is satisfied.

Tuple space languages are concurrent languages, that foster the definition of autonomous entities of computation (the processes), and offer mechanisms for their synchronization and communication. In particular, they represent one of the most acknowledged models of *coordination*. Tuple space languages are based on the idea that a dynamic collection of tuples can act as shared state of concurrent processes, and play the role of coordination media among them.

To build a refinement calculus for tuple spaces, we address three points, in this paper:

(1) We single out a specification language, a variation of first-order temporal logic. Temporal relations between propositional formulae are not expressive enough to describe relations between tuple spaces, which are multisets of atoms. The specification language, called Oikos-tl, includes three new temporal operators that enhance the expressive power of the logic, permitting to directly link state transitions and state configurations. The semantics of the specification language is formally defined, and a set of useful properties for refinement are shown.

(2) We introduce a reference language for tuple spaces, dubbed TuSpReL, and define its axiomatic and operational semantics. We need the former to derive properties, the latter to describe the allowed computations of a system. We relate these descriptions, and guarantee that using the axiomatic semantics we can derive properties, which are correct and complete with respect to the operational behavior. The non-deterministic features of tuple space languages make this result new, and more complex than in other programming paradigms. One of the contributions of our work is the idea to derive weakest preconditions exploiting the *demonic strict* choice in non-deterministic selection. The transition system defining the operational semantics is based on the new notion of enabling precondition, which exploits the *angelic strict* choice.

* Corresponding author. E-mail: semini@di.univpi.it

PII: S0167-6423(98)00021-5

(3) To build the refinement calculus, we take a compositional approach. We first consider the basic statements of the language, and say under which conditions they satisfy a property, then compose these proofs to derive that a system refines a specification. Finally, in the refinement calculus definition, we extend to tuple space languages the ability to exploit logic formulae to specify the behavior of unrefined modules: in the intermediate steps, a system is only partially written in the programming language, and the still unrefined features are described by logical formulae.

Keywords: Coordination models; Tuple spaces; Refinement calculus; Program specification

1. Introduction

We discuss here the key features of tuple space languages, the motivations of our work, and the state of the art.

1.1. Framework

A coordination model is an interaction model between concurrent processes that communicate via a shared media, i.e., an interaction model for the components of a distributed system. The current large growth of local area networks is bringing the attention on distributed architectures, and coordination languages, which provide a suitable communication model for distributed entities, have become popular in the last years [27, 31]. One of the most acknowledged models of *coordination* is related to *tuple space languages*: these concurrent languages foster the definition of autonomous entities of computation (the processes), and offer mechanisms for their synchronization and communication. Tuple space languages are based on the idea that a dynamic collection of tuples can act as shared state of concurrent processes, and play the role of coordination media among them. The operations to read and change a tuple space, represent the primitive statements of these languages. They exploit pattern matching or unification. As a consequence, processes access the state values by matching the state content, rather that exploiting variable addresses. This feature is known as *associative* access to data and strongly characterizes the paradigm. Indeed, associative access is an aspect of declarativity.

The tuple space paradigm directly addresses the implementation of coordination policies: *separation* [21] of the computation and communication facets of a process, and *uncoupling* [20] between concurrent processes, are the key features towards this aim.

Separation is achieved thanks to a suitable syntactic structuring of programs: process descriptions consist of a communication component, i.e. the read–write operations on the shared state, and a separate computation component, a program written in any programming language.

Uncoupling enhances modularity: no interaction between different processes must be explicitly described by the programmer. In particular, a process does not have to know

the identity of the processes with which it cooperates, as it happens in the message passing paradigm, nor to deal with explicit synchronization and mutual exclusion constructs, as required in a paradigm based on shared variables. Communication via the tuple space is asynchronous, and needs no explicit addressing, thus fully satisfying the uncoupling requests. The processes involved in a computation can freely work both in parallel (along the time dimension) and in concurrency (along the space dimension), accessing the same store [20]. Actually, a tuple is a convenient way to communicate: the presence of a tuple in the tuple space represents a signal; at the same time, tuples carry values.

Besides, tuple space languages have declarative facets. Declarativity is an advantage for the programmer, and we believe that this property is most useful when the programming goal is to build a concurrent application. In these cases, the declarative approach allows to describe events and reactions, while the procedural alternative requires to explicitly receive external inputs, test complex multiple conditions, and produce proper outputs.

Actually, the combination of tuple spaces with logic programming yields a paradigm that well integrates cooperation and declarativity. The expressive power of a tuple space language is enhanced by unification: process descriptions can include complex conditions on the tuple space, yet expressed in a very concise manner. The cooperation and the computation facets of a process (the interaction with the tuple space and the computation fired by the specified configurations of the tuple space, respectively) can be uniformly described by the same formalism.

The tuple space paradigm inspired the design of many programming languages, like Linda [20, 22], Swarm [51], LO [7], Γ [12, 13], Tao [50, 46], manifold [9], SP [16], ESP [19], PoliS [26], Pâté [6], *eta* [4, 5], Multi-Prolog [14], and μ^2-Log [15]. A focussed discussion of the characteristics of some of these languages can be found at the beginning of Section 2.

1.2. Motivations

Our target is the definition of a *logical*, *heterogeneous*, and *compositional* refinement calculus for tuple space languages. We fix the meaning of these attributes, and discuss why they are important for a refinement calculus. In the next section, we discuss in this perspective the recent works on refinement in tuple spaces, and outline the possibilities for improvement.

First, the *logical* nature of a refinement is discussed, and compared with the *semantic* one. In the logical approach we say that $P' \rhd P$ (P' refines P) if, given an appropriate set S of properties (logic formulae), $\forall f \in S : P'$ satisfies f whenever P satisfies f. As an example a sequential sorting algorithm, satisfying the property that the output list is correctly sorted, is refined by a parallel algorithm satisfying the same property. In the semantic approach, on the contrary, we say that $P' \geqslant P$ if the semantics of P is an appropriate abstraction of the semantics of P'. As an example take the trace semantics of an algorithm A, and of a more detailed version of it, A', that adds some

new information. We can say that the latter refines the former if we can find a suitable observation function that reduces the semantics of A' to be equivalent to that of A. The difference is methodological, in the first case results are based on the static verification of logical properties, while in the second case they are based on an analysis of the semantic domain.

A very general logical framework is provided by Hoare [35, 36], who claims that "programs are predicates". Any program P is identified with the strongest predicate describing its behavior, and a calculus for deriving program predicates from basic statement predicates is proposed. The refinement relation then collapses to logical implication.

Morgan's approach [47, 48] is logical as well. It is based on the weakest precondition semantics, and the refinement relation is defined as follows: given two programs P and P', P' refines P if and only if for all formulae f : $wp(P, f) \rightarrow wp(P', f)$.

On the semantic side, Broy's approach is worth considering, and deals with the refinement of networks of interactive modules [17, 18]. A functional specification technique is used to supply the semantics of each module: a specification is a set of properties that the I/O behavior must satisfy, the semantics is the set of I/O functions satisfying the specification. The refinement of a module specification P is a specification P' with $P' \rightarrow P$. Note that $P' \rightarrow P$ if and only if for every I/O function f : $P'(f) \rightarrow P(f)$.

We prefer the logical approach, because we can provide a general framework in which we can discuss a refinement relation between programs, as well as a satisfaction relation of a program with respect to a specification given in a logical language. On the contrary the semantic approach only allows to compare pairs of programs. Besides, the choice of focusing on the I/O behavior is limiting, since it does not allow, for instance, to express properties of the computations, like safety or liveness, that are far-reaching when describing concurrent systems.

Let us now discuss what we mean by *heterogeneity*. This property deals with the intermediate steps of a step-wise derivation, the *unrefined programs*, which can either be written in a programming language (e.g., the language of the final program), or in a different non-executable (e.g., specification) language. In the first case we have the advantage that we can execute and test the unrefined programs, while in the second case we are discharged from considering inessential, operational details. The heterogeneous approach belongs to the second case: each intermediate step is partially written in a programming language, and some unrefined features are described by logical formulae. In other words, we allow to exploit logic formulae to specify the behavior of unrefined programs as, for instance, in [1]. In our opinion, this solution well meets the idea of step-wise refinement: the designer can, for instance, concentrate on a part of the program assuming that the others behave correctly.

Finally, we take into account *compositionality*. In the context of refinement, compositionality allows to derive a refinement relation between complex systems from a refinement relation between sub-components. Going back to the example seen above, in Hoare's works [35, 36], the monotonicity results about statement compositions with

respect to the refinement relation support modular (compositional) refinement. Broy's approach is compositional as well: composition operators (including parallel composition) allow to derive the behavior of complex systems from the semantics of their modules, and, enjoying a monotonicity property with respect to the refinement relation, support a compositional refinement.

It is actually self-evident that a compositional approach is more desirable than a non-compositional one, in particular when dealing with the development of systems built by a large number of different components, and we need not to motivate further our preference.

1.3. State of the art

In her doctoral thesis [24, 25], XiaoJun Chen presents a formalism for the verification of multiple tuple space systems, built by stepwise refinement, namely ESP systems [19]. An ESP system consists of two kinds of active components: a set of agents and a set of subsystems having the same structure as an ESP system. A subsystem is treated as a system of which we have only a black-box view. The semantics of an ESP system is given in terms of labeled transition systems. Chen's approach to refinement can be characterized as semantic: a program P' refines a program P if the semantics of P is an appropriate abstraction of the semantics of P'. We contend here that an approach based on logic is better for system development.

We find an example of logical approach to refinement in tuple spaces in the work on Swarm [28, 52]. Their derivation strategy starts with a specification of the program, which describes initial and final states, and some global properties that must hold during program execution. The initial specification is refined, producing a very detailed version that can be directly mapped into a program. Correctness of the derivation steps is proved by applying the *Swarm proof logic* [28], an assertional logic similar to that of Unity [23]. The building blocks of their proofs are assertions of the kind $\{p\}\,R\,\{q\}$, with R rewriting rule. However, they lack a formal derivation for these assertions. The difficulty mainly lies in the non-deterministic selection of the tuples made by the matching of the rule guard with the tuple space. The definition of a formal method to prove $\{p\}\,R\,\{q\}$, allowing this gap to be closed, represents one of the main results of this paper (see Section 4).

A similar approach has been taken by Goeman et al. with the definition of the ImpUnity framework [34]. The ImpUnity logic permits to reason on coordination programs. These are written in ImpUnity, a programming language based on Unity, action systems, and on the tuple space model of communication. The language permits to declare private variables in programs, and these declarations are used to refine the programs compositionally. As for Swarm and for our approach, program properties are derived using assertion on the form of Hoare triples, but no formal definition of the axiomatic semantics is given.

Compositionality results for programs based on tuple spaces have been proposed also by Jacquet and De Bosschere in [37]. They discuss two different methodologies for

composition. The first one permits to derive the operational semantics of a program, a set of computations, by composing the semantics of each statement (a goal), given in terms of traces and substitutions. The second methodology is based on a more abstract, logical approach. Program properties are expressed in Unity, properties of the composition of programs are obtained with a rely-guarantee reasoning similar to the one proposed in [1]. However, the problem of verifying the (basic) components is not addressed, and no notion of heterogeneous system is considered.

1.4. Structure of the paper

In Section 2 we define TuSpReL, a reference tuple space language and provide an example program. In TuSpReL, processes are reactive *agents* described in terms of *rules*, where a rule defines the interaction protocol with the tuple space: the *guard* defines conditions on the tuple space (firing conditions); the *body* starts an inner Prolog computation; and a *tell* defines a write operation on the tuple space.

Our refinement calculus is based on a temporal logic oriented specification language, presented in Section 3. In our tuple spaces, a state is composed of ground atoms and a computation is a sequence of states: state formulae talk about the presence (or absence) of some atoms, general formulae, integrating state formulae and temporal operators, describe computations. In state formulae we capture the fact that states are *multisets* of atoms by introducing a non-idempotent conjunction. We inherit Unity logic [23] and introduce three new temporal operators, which directly relate the *appearance* (instead of generic presence) of a tuple to other conditions of the current state of a computation. We refer to this specification language as Oikos-tl, since it has been inspired by the experience with previous developments in the Oikos project [3, 43]. Due to the new operators, Oikos-tl is well suited to deal with events in reactive systems.

In Section 4 we define both an axiomatic and an operational semantics for our prototype language. The axiomatic semantics is in the weakest precondition style, and the transition system defining the operational semantics is based on the new notion of enabling precondition, which exploits the angelic choice in non-deterministic selection. We relate operational and weakest precondition semantics so that the properties derived using the axiomatic semantics are correct and complete with respect to the operational behavior. This result overcomes the problems related to the basic computational statement, which is a non-deterministic rule rather than a simple deterministic assignment.

In Section 5 we show how to refine Oikos-tl specifications into TuSpReL programs. We first deal with safety properties, showing when a rule refines a given property, and how to compose these proofs to derive the more general relation that a system refines a set of safety constraints. We then treat liveness. In the second part of the section, we introduce heterogeneous systems and propose a methodology of refinement.

Finally, in Section 6, we provide an example of refinement, the problem addressed being Eratosthenes' sieve.

2. TuSpReL, a reference language

In most tuple space languages,[1] processes are reactive *agents* described in terms of *rules*. We are interested, in this work, in the inspection of these languages. A rule defines the interaction protocol with the tuple space and can be sketched in the following way: a guard defines conditions on the tuple space (firing conditions); a query starts an inner computation in a given calculus; and a tell defines a write operation on the tuple space. The inner computation is functional and affects the tuple space in terms of values assigned to the arguments of the tell. Linda is a notable exception: the processes have their own state and flow of control and exploit the tuple space only for coordination purposes. An atomic interaction with the tuple space involves only one tuple, i.e., each interaction consists in the read, consume, or write of a single tuple.

We fix here a prototype language based on tuple spaces and rules, as a common subset of the languages of this paradigm, that will act as reference language in the rest of the work. Formal syntax is provided in Table 1. For the sake of reference, we call it TuSpReL, for Tuple Space Reference Language.

A tuple space consists of a multiset of ground atom. We do not see any difficulty in generalizing to terms, to accommodate languages as defined in [37]. A program is composed of set of rules. Optionally, it can include a Prolog program and a partial order between rules, expressing priorities. We assume the number of rules of a system to be invariant during system execution: no new rule is activated at run time. Rules are composed by a guard, a commit operator, an optional body, and a tell, and take the form

GUARD | BODY. TELL

The GUARD is a sequence of *ask* and *consume*: the formers are queries on the contents of the tuple space, the latters indicate the set of atoms that must be removed from it. Moreover, the guard can contain some Prolog primitives, e.g. $X > Y, X \neq Y$. These are used to express relations between the variables occurring in the guard: the variables in these Prolog queries must also occur in queries or consume conditions on the contents of the tuple space in the same guard. Ask, consume, and write operations are performed exploiting unification. An ask condition can take the form A as well as **not** A; in the first case it is satisfied if and only if the tuple space contains an atom B unifying with A, while in the second case the tuple space must not contain any instance of A. The BODY is an (optional) Prolog atomic goal, evaluated with respect to the Prolog program. TELL is a sequence of atoms and specifies a write operation. It can be partitioned in two parts: the first part (or the unique one if no partition exists) is executed in the case of successful evaluation of the BODY, or if the BODY is empty. The second part is considered in the case of finite failure during BODY evaluation. We require that all the variables appearing in the TELL are instantiated when the tell operation is made, since we want only ground atoms in the tuple space. A necessary condition to satisfy this constraint is

[1] Namely, Swarm, Γ, SP, ESP, PoliS, Pâté, *eta*, and Tao.

Table 1
TuSpReL syntax

PROGRAM	::=	RULES BODY_DEFINITION PRIORITIES
RULES	::=	LABEL : RULE | RULES RULES
RULE	::=	GUARD | POST % | is the commit operator
GUARD	::=	_ | CONS GUARD | ASK GUARD
CONS	::=	**cons** ATOM
ASK	::=	**ask** ATOM | **ask not** ATOM | **ask** TEST
TEST	::=	X = Y | X ≠ Y | ...
POST	::=	BODY TELL; TELL | BODY TELL | TELL
BODY	::=	**body** ATOM.
TELL	::=	_ | **tell** ATOM | TELL TELL
BODY_DEFINITION	::=	_ | **with** *prolog program*
PRIORITIES	::=	_ | **and** ORDER
ORDER	::=	LABEL > LABEL | ORDER, ORDER
LABEL	::=	*ground term*
ATOM	::=	*prolog atom*
X, Y	::=	*terms*

Note. Syntactic categories are capitalized. The symbols ::= and | belong to the BNF notation, and _ denotes the empty choice. The symbols ; . : >, and \| belong to the language, as well as the Prolog primitives =, ==, ≠, etc. Keywords and primitives of the language are in boldface. The former set includes: **with, ask, cons, body**, and **tell**; the latter includes **not**.

A rule is composed of two parts separated by the commit operator "|". The first part, GUARD, is the rule guard. The second, POST, is a conjunction of the body and the tell of the rule. They may be empty, however a rule with an empty guard is usually undesired, since it would be executed continuously. Both ASK and CONS are sequences of tagged atoms. The ask guard (ASK) may contain primitives derived from Prolog (TEST), to express relations between the variables appearing the guard. POST consists of a body and a tell. The body (BODY) is a Prolog atomic atom. TELL is a sequence of write declarations. It can be composed of a unique part or of two parts, separated by ";".

that the variables occurring in the TELL also appear in the body or in a positive literal of the GUARD. Actually, in some tuple space languages this constraint is not imposed. However, the semantic definition would be much more complex in this case.

We will define the semantics of our language formally in Section 4. We provide here a very simple example to give an intuitive idea of rule behavior. The rule

cons $d(\mathrm{X})$ **ask** $\mathrm{X} > 0$ | **body** Y *is* $\mathrm{X} + 1$. **tell** $d(\mathrm{Y})$

looks for an atom matching $d(\mathrm{X})$ with $\mathrm{X} > 0$. If no such atom is in the tuple space, the rule is not executed. Otherwise, the rule can be fired, i.e. the commit operator " | " is taken. One of the matching atoms, say $d(c)$, is non-deterministically selected and consumed (removed from the tuple space). The body calculates $c+1$, call c' the result, tell writes $d(c')$ in the tuple space.

Table 2
Example in the prototype language: Eratosthenes' Sieve

```
R1 : cons max(M)
     ask primes(P)   ask M < P
   | body M' is M + 1.
     tell max(M')    tell n(M')

R2 : cons n(X)      ask n(Y)
     ask Y < X      ask X mod Y = 0
   |

R3 : ask not done
   |
     tell done

and R1 > R3, R2 > R3
```

Guard evaluation is an atomic operation and is independent on the order in which *ask* and *consume* conditions appear.[2] Informally, the whole guard is first evaluated considering all the conditions in it as *ask* conditions, then the atoms specified by the *consume* conditions are removed. For instance, in state $\{p(a)\}$ the evaluation of the following guards succeeds:

ask $p(\mathrm{X})$ **cons** $p(\mathrm{X})$	**ask** $p(\mathrm{X})$ **ask** $p(\mathrm{X})$
ask $p(\mathrm{X})$ **cons** $p(\mathrm{Y})$	**ask** $p(\mathrm{X})$ **ask** $p(\mathrm{Y})$ **ask** $p(\mathrm{Z})$
cons $p(\mathrm{X})$ **ask** $p(\mathrm{Y})$	**cons** $p(\mathrm{X})$ **ask** $p(\mathrm{X})$

while the evaluation of **cons** $p(\mathrm{X})$ **cons** $p(\mathrm{Y})$ and of **ask** $p(\mathrm{X})$ **ask** $p(\mathrm{Y})$ **ask** $\mathrm{X} \neq \mathrm{Y}$ fails.

In Table 2, we show an example program in the prototype language. The problem addressed is Eratosthenes' Sieve: the program produces all tthe prime numbers minor than a given number P, provided that the tuples *primes*(P), *max*(2), and $n(2)$ are in the space. The solution is a non-trivial example of cooperation among a producer and a consumer. Rule R_1 acts as number generator, and generates orderly the numbers less than P; rule R_2 acts as sieve; rule R_3 states the end of the computation. The priority declaration $R_1 > R_3$ guarantees that the computation is not terminated before all the natural numbers less than P have been generated; $R_2 > R_3$ guarantees that the computation is not ended if the sieve can still detect and remove some non-prime numbers. Note that the sieve and the number generator can behave concurrently. To this purpose, we introduced *max*, playing the role of a counter, instead of having a generator simply asking for the presence of $n(\mathrm{X})$, with $\mathrm{X} < \mathrm{p}$, and the absence of $n(\mathrm{X}+1)$ to produce $n(\mathrm{X}+1)$. In fact, in this simplified solution, the generator would continue to produce the numbers that are removed by the sieve.

[2] However, the praxis is to list the binding conditions first.

3. Oikos-tl, a specification language

Our refinement calculus is based on a temporal logic oriented specification language, Oikos-tl. We first discuss the main differences between Oikos-tl and the best known temporal logics, and then we supply syntax and semantics.

(1) *Non-idempotent conjunction.* In most tuple space languages, the states of a computation are multisets, hence neither propositional nor first order logic classical languages are well suited to describe their properties. Programming with multisets as primitive data structure is often useful, in particular when prototyping: it allows to model concisely and naturally the sharing of resources that can appear with multiple occurrences. Multiset union is not-idempotent and we want to reflect this property in the logic framework: we integrate temporal operators and a non-idempotent connective $+$ that resembles the $\otimes$ (*times*) connective of linear logic [33]. We contend that the resulting logic is more appealing to software developers. Besides, it turns out that our non-idempotent conjunction is a useful operator also when specifying tuple space systems like Swarm systems [51], where the tuple space is a *set* of tuples. It is fairly common to need to constrain the computation in such a way that no two atoms with the same predicate p appear in the tuple space at the same time. To specify this constraint, on a *set* of tuples, with the classical $\wedge$ we would have to write something like

$$\Box \sim [(p(x_1, \ldots, x_n) \wedge p(y_1, \ldots, y_n)) \wedge (x_1 \neq y_1 \vee \cdots \vee x_n \neq y_n)]$$

while

$$\Box \sim (p(x_1, \ldots, x_n) + p(y_1, \ldots, y_n))$$

is all we have to say if we can use a non-idempotent conjunction. Since we are discussing here the use of operator $+$ to describe properties of sets, the meaning of the two formulae is the same: we cannot have two identical atoms in a set.

(2) *No explicit quantification.* Still addressing readability in specifications, we want to avoid explicit quantifications in the formulae, miming the simplicity of logic programming. Then, a formula is interpreted as universally or existentially quantified, as stated by the semantics given in Section 3.2.

The assumption is that, in almost all cases, there is a natural quantification of the variables appearing in a formula. For instance, the intuitive meaning of $p(x) \rightarrow \Diamond q(x, y)$ is $\forall x\, [p(x) \rightarrow \Diamond \exists y\, q(x, y)]$.

(3) *New operators.* We inherit Unity logic [23] and introduce three new temporal operators: AFTER, NEEDS, and CAUSES. Unity operators, like UNLESS, ENSURES, and LEADS-TO, are sometimes too strong in the very first refinement steps. In addition, they are not suited to directly relate the *appearance* (instead of generic presence) of a tuple to other conditions of the current state of a computation. The new operators enhance the expressive power of temporal logic when describing reactive systems as the ones we are taking into consideration, leading to more readable and concise specifications. In particular, they allow us to specify straightforwardly conditions

relating state changes (transitions) to state configurations or to further reactions, as loosely as it is often needed in the very first refinement steps. Most of the time we want to say something like: "if p appears, then ..." without making choices depending on the fact that p would remain true or not. For instance, Unity's $p \mapsto q$ is stronger than "if p appears, then q will be true" (see Section 3.3).

(4) *Variable and predicate interpretation.* In tuple spaces, a state is composed of ground atoms and a computation is a sequence of states ruled by reactions to the state contents. Hence, it is natural that a formula describing a state (*state formula*) talks about the presence (or absence) of some atoms, and that a formula describing a computation relates state formulae by temporal operators. For instance, $\Box(p(\mathrm{x}) \rightarrow \Diamond q(\mathrm{x}))$ says that every state containing an atom $p(\mathrm{x})$ is eventually followed by a state containing $q(\mathrm{x})$.

According to logic programming [40], we define the interpretation of a predicate p in a state s as the set of tuples of terms $t_1, \ldots, t_n$ such that the atom $p(t_1, \ldots, t_n)$ belongs to s. This set clearly changes during a computation and, consequently, the interpretation of a predicate can change from a state to the next. On the contrary, in the literature [39, 42, 41, 49], predicates have a fixed interpretation.

Moreover, in the literature, the set of variables appearing in a formula is split in two subsets: *local* and *global* variables (called also flexible and rigid, respectively, by Manna and Pnueli [42]). Global variables have a fixed interpretation in the states of a computation. Local variables, on the contrary, can be interpreted differently in different states. For instance, in

$$\Box\forall v\,[(x = v) \rightarrow \Diamond(x = 2v)]$$

x is a local variable, v a global one. In our proposal, we do not distinguish between local and global variables, all variables being considered as global. This is a consequence of the computational model. In tuple space languages there are no identifiers, while the need for local variables is tightly related to identifiers, whose values change during a computation.

In the next section we describe state formulae, in Section 3.2 we extend state formulae with temporal operators, and, in Section 3.3, we introduce the non-standard temporal operators and some theorems for them.

3.1. State formulae logic

A state formula represents an assertion on a program state: it specifies conditions on the presence, or absence, of some tuples (atoms in our setting). To define them, we fix two distinguished sets of predicate symbols: *Pred* and *Test*. The first one represents the set of predicates appearing in the states of a computation, while the second contains predicates inherited from Prolog primitives ($\neq$, $=$, *is* , ...).[3]

[3] In the following section we will use the prefix notation for these predicates too.

Table 3
Success substitutions

A	s	$pos(A)$	$\Sigma = \{\vartheta \mid s \models_w A\}$	$\Sigma' = \{\vartheta \mid s \models A\}$
$\sim p(\mathrm{X})$	$\{q(a)\}$	$\emptyset$	$\{\varepsilon\}$	$\{\varepsilon\}$
$\sim p(\mathrm{X})$	$\{p(a)\}$	$\emptyset$	$\emptyset$	$\emptyset$
$\sim(p(\mathrm{X}) + p(\mathrm{X}))$	$\{p(a), p(b)\}$	$\emptyset$	$\{\varepsilon\}$	$\{\varepsilon\}$
$p(\mathrm{X}) + \sim q(\mathrm{Y})$	$\{p(a), q(b)\}$	$\{\mathrm{X}\}$	$\{(\mathrm{X}/a, \mathrm{Y}/a)\}$	$\emptyset$
$p(\mathrm{X}) + \sim q(\mathrm{X})$	$\{p(a), q(b)\}$	$\{\mathrm{X}\}$	$\{(\mathrm{X}/a)\}$	$\{(\mathrm{X}/a)\}$
$p(\mathrm{X}) + \sim(q(\mathrm{Y}) + \mathrm{X} \neq \mathrm{Y})$	$\{p(a), q(a)\}$	$\{\mathrm{X}\}$	$\{(\mathrm{X}/a)\}$	$\{(\mathrm{X}/a)\}$

The semantic definition for these formulae is given in two steps: we provide a weaker notion of validity, which can be given in an inductive way, then we base the semantics on it (Definition 4).

Definition 1 (*State formulae*).
- An atom $p(\bar{\mathrm{X}})$ with $p \in Pred$ or an atom $t(\bar{\mathrm{X}})$ with $t \in Test$ are *positive* formulae.
- If A and B are positive formulae, then so is $A + B$.
- If A is a positive formula then $\sim A$ is a *negative* formula.
- If A is a positive or negative formula, then it is a *state* formula.
- If A and B are state formulae, then so is $A + B$.

We need to introduce a normal form. For instance, $p(\mathrm{X}) + \mathrm{Y} > \mathrm{X} + \sim q(\mathrm{Y})$ is changed in $p(\mathrm{X}) + \sim(q(\mathrm{Y}) + \mathrm{Y} > \mathrm{X})$, which has the same intended meaning. Normalization simplifies the semantic definition: it would be uselessly complex to give a semantics which does not distinguish among the formulae above. In particular, we want variables in *Test* predicates to be ground when evaluated.

Definition 2 (*State formulae normal form*). State formula A:

$$p_1(\bar{\mathrm{X}}) + \cdots + p_n(\bar{\mathrm{Y}}) + t_1(\bar{\mathrm{Z}}) + \cdots + t_m(\bar{\mathrm{W}}) + \sim B_1 + \cdots + \sim B_l$$

is in normal form if the variables occurring in $t_1(\bar{\mathrm{Z}}), \ldots, t_m(\bar{\mathrm{W}})$ also to occur in $p_1(\bar{\mathrm{X}}), \ldots, p_n(\bar{\mathrm{Y}})$. We call $pos(A)$ the set of variables in $p_1(\bar{\mathrm{X}}), \ldots, p_n(\bar{\mathrm{Y}})$.

Example 3. A couple of examples of state formulae follow. Other examples are in Table 3.

$$A = p(\mathrm{X}) + q(\mathrm{Y}) + \mathrm{Y} > \mathrm{X} + \sim(q(\mathrm{Z}) + \mathrm{Z} < \mathrm{X}), \qquad pos(A) = \{\mathrm{X}, \mathrm{Y}\}$$

$$B = p(\mathrm{X}) + \sim(q(\mathrm{Y}) + q(\mathrm{Z}) + \mathrm{X}\ is\ \mathrm{Y} - \mathrm{Z}), \qquad pos(B) = \{\mathrm{X}\}$$

Definition 1 excludes general formulae with nested negations, like, for instance $\sim(A + \sim(A + B))$: in that case, both the semantic definition and the decision algorithm, which permits to decide if a multiset is a model for a state formula [53], would be uselessly complex.

Interpretations for state formulae are tuple spaces, i.e. multisets of ground atoms. Given an interpretation s and state formula A, we define $s \models \langle A, \vartheta \rangle$, to be read: A holds in s with substitution ϑ, or equivalently s is a model for A with success substitution ϑ. In the following definition, label "w" stays for *weak*; the stronger notion of validity, that we require, constrains substitution ϑ to bind only the variables that occur in the positive sub-formula of A.

Definition 4 (*Semantics*: $s \models \langle A, \vartheta \rangle$). We say that a state s models a formula A (with substitution ϑ), written $s \models \langle A, \vartheta \rangle$ if and only if

$$s \models_{\mathrm{w}} \langle A, \vartheta \rangle \quad \text{and} \quad \vartheta_{|\, pos(A)} = \vartheta$$

where $\vartheta_{|\, B}$ is the restriction of ϑ to the variables occurring in B; $\uplus$ is multiset union:

$$s \models_{\mathrm{w}} \langle p(\bar{\mathrm{x}}), \vartheta \rangle \quad \text{iff} \quad \begin{cases} p(\bar{\mathrm{x}})\vartheta \in s & \text{if } p \in \mathit{Pred}, \\ p(\bar{\mathrm{x}})\vartheta = \mathit{true} & \text{if } p \in \mathit{Test}, \end{cases}$$

$$s \models_{\mathrm{w}} \langle \sim A, \vartheta \rangle \quad \text{iff} \quad \forall \vartheta' \text{ not } s \models_{\mathrm{w}} \langle A, \vartheta' \circ \vartheta \rangle,$$

$$s \models_{\mathrm{w}} \langle A+B, \vartheta \rangle \quad \text{iff} \quad \begin{cases} s \models_{\mathrm{w}} \langle A, \vartheta \rangle, \\ s \models_{\mathrm{w}} \langle B, \vartheta \rangle \\ \exists s_1\, s_2 : \; s_1 \uplus s_2 \subseteq s, \; s_1 \models_{\mathrm{w}} \langle A, \vartheta \rangle, \; s_2 \models_{\mathrm{w}} \langle B, \vartheta \rangle. \end{cases}$$

In other words, $p(\bar{\mathrm{x}})$ holds in a state s with substitution ϑ if and only if s contains the (ground) atom $p(\bar{\mathrm{x}})\vartheta$, for $p \in \mathit{Pred}$, or the (ground) atom $p(\bar{\mathrm{x}})\vartheta$ is true in the usual Prolog semantics [54] for $p \in \mathit{Test}$. A state s is a w-model for $\sim (A)$ with substitution ϑ if and only if it does not exist any (grounding) substitution ϑ' such that s is a w-model for A with substitution $\vartheta' \circ \vartheta$ (in $\vartheta' \circ \vartheta$, ϑ is applied first).

Finally, the sum. State s w-models $A+B$ if and only if: it w-models A; it w-models B; it can be partitioned in two sub-states w-modeling A and B, respectively. The first two conditions are mandatory, otherwise $a + \sim a$ would have a w-model ($\{a\} = \{a\} \uplus \{\,\}$ with $\{a\}$ w-modeling a and $\{\,\}$ w-modeling $\sim a$) while we want it to be false. Moreover, the third condition cannot be skipped: consider the formula $a + a$, we want a to appear at least twice in a w-model for it, while the first two conditions alone would allow $\{a\}$ to be a w-model. Some examples follow. Some more examples are in Table 3.

Example 5.

- Let a be a propositional atomic formula. Then $\{a\} \models \langle a + \sim(a + a), \varepsilon \rangle$. In fact, both a and $\sim(a + a)$ hold in $\{a\}$. Moreover, $\{a\} = \{a\} \uplus \{\,\}$ with $\{\,\}$ model of $\sim(a + a)$.
- $\{\,\}$ models every negative formula.
- $\{p(a), p(b)\} \models \langle p(\mathrm{X}) + p(\mathrm{Y}) + \sim[p(\mathrm{Z}) + p(\mathrm{Z})], \vartheta \rangle$, with $\vartheta = \{\mathrm{X}/a, \mathrm{Y}/b\}$ or $\vartheta = \{\mathrm{X}/b, \mathrm{Y}/a\}$.
- $\{p(1), p(3)\} \models \langle p(\mathrm{X}) + p(\mathrm{Y}) + \mathrm{X} > \mathrm{Y}, \{\mathrm{X}/3, \mathrm{Y}/1\} \rangle$.

Example 6. This example shows the difference between semantics and weak semantics:

$$\{p(1), p(5), q(3)\} \begin{matrix} \models^{\mathrm{w}} \\ \not\models \end{matrix} \langle p(\mathrm{x}) + \sim(q(\mathrm{y}) + \mathrm{x} \geqslant \mathrm{y}),\ (\mathrm{x}/5, \mathrm{y}/4) \rangle$$

In fact, substitution $(\mathrm{x}/5, \mathrm{y}/4)$ also binds a variable not appearing in the positive subformula.

We did not find a complete proof system for our state logic. This is not surprising, since it is a kind of dual to the problem of finding a model for linear logic. In particular, in our case, the substitution principle does not hold. In fact, let a and b be propositional atomic formulae, $\vdash$ any derivation relation of a sequent calculus sound with respect to the semantics given in Definition 4, then we would have

$$a + \sim(a + a) \not\vdash \sim a$$
$$a + \sim(a + b) \vdash \sim b$$
$$a + \sim(a + a + a) \not\vdash \sim(a + a)$$

However, in [53], we describe an algorithm to decide if a multiset is a model for a state formula.

3.2. Adding temporal operators

We add temporal operators to relate state formulae. In this section, we introduce a subscript σ to distinguish between the operators we use (with implicit quantification) and normal temporal operators.[4] The relation between them is given in Definition 10, and defines the way to make the quantification explicit. We inductively construct σ-formulae out of state formulae.

Definition 7 (*Well formed σ-formulae*). If A is a state formula, then A is a σ-wff. If p and q are *σ-wff*, then so are $\neg_\sigma p$, $p \rightarrow_\sigma q$, $\Box_\sigma p$, $\Diamond_\sigma p$, $\bigcirc_\sigma p$, $p \wedge_\sigma q$.

In σ-formulae only some of the usual equivalencies are true, because of the implicit quantification. For instance, $\Diamond_\sigma$ cannot be defined in terms of $\Box_\sigma$ and is thus introduced explicitly. Some other operators are derived.

Definition 8 (*Derived operators*).

$$\begin{aligned}
p \vee_\sigma q &= \neg_\sigma(\neg_\sigma p \wedge_\sigma \neg_\sigma q) \\
p \mapsto_\sigma q &= \Box_\sigma(p \rightarrow_\sigma \Diamond_\sigma q) \\
p \,\mathrm{UNLESS}_\sigma\, q &= \Box_\sigma((p \wedge_\sigma \neg_\sigma q) \rightarrow_\sigma \bigcirc_\sigma(p \vee_\sigma q)) \\
\mathrm{STABLE}_\sigma\, p &= p \,\mathrm{UNLESS}_\sigma\, \mathit{false} \\
p \,\mathrm{UNTIL}_\sigma\, q &= (p \,\mathrm{UNLESS}_\sigma\, q) \wedge_\sigma p \mapsto_\sigma q \\
\mathrm{INIT}_\sigma\, p &= p
\end{aligned}$$

[4] In the next sections we will deal only with σ-formulae, and simplify the notation dropping label σ.

Remark 9. $\text{INIT}_\sigma\ p$ means that p holds in the initial state. The equivalence above follows from Definition 14. We explicitly use INIT for clarity.

We define a function t on σ-formulae, which makes quantification over variables explicit and is needed to define σ-formulae semantics. In the definition, we abbreviate with $\exists\vartheta_p$ the sentence: "$\exists\vartheta$, ϑ is restricted to the variables occurring in p" (similar for $\forall$), while with $\exists\vartheta_{p_{\setminus ex}}$ we mean: "$\exists\vartheta$, ϑ is restricted to the variables occurring in p and not already bound by an external substitution". For instance, in $\forall\vartheta_{p\cap q}\,[p(x)\vartheta \rightarrow \Diamond\,\exists\vartheta'_{q_{\setminus ex}}\,\langle q(x,y)\vartheta,\vartheta'\rangle]$, ϑ is a substitution for x, and ϑ' a substitution for y.

Definition 10 (*Translation function from σ-formulae to temporal logic formulae*).

$$
\begin{aligned}
t(A) &= \exists\vartheta_{A_{\setminus ex}}\langle A,\vartheta\rangle \qquad (A \text{ state formula})\\
t(\neg_\sigma\, p) &= \forall\vartheta_p\,\neg\langle t(p),\vartheta\rangle\\
t(p \rightarrow_\sigma q) &= \forall\vartheta_{p\cap q}\,[t(p\vartheta) \rightarrow t(q\vartheta)]\\
t(\Box_\sigma p) &= \exists\vartheta_{p_{\setminus ex}}\ \Box\,\langle t(p),\vartheta\rangle\\
t(\Diamond_\sigma p) &= \exists\vartheta_{p_{\setminus ex}}\ \Diamond\,\langle t(p),\vartheta\rangle\\
&= \Diamond\,\exists\vartheta_{p_{\setminus ex}}\,\langle t(p),\vartheta\rangle\\
t(\bigcirc_\sigma p) &= \exists\vartheta_{p_{\setminus ex}}\bigcirc\langle t(p),\vartheta\rangle\\
&= \bigcirc\,\exists\vartheta_{p_{\setminus ex}}\langle t(p),\vartheta\rangle\\
t(p \wedge_\sigma q) &= \exists\vartheta_{(p\cup q)_{\setminus ex}}\,[\langle t(p),\vartheta\rangle \wedge \langle t(q),\vartheta\rangle]
\end{aligned}
$$

Example 11.

$$
\begin{aligned}
t(\Box_\sigma p) &= \exists\vartheta_p\ \Box\,\langle p,\vartheta\rangle\\
t(\neg_\sigma\ \Diamond_\sigma\ \neg_\sigma p) &= \neg\ \Diamond\ \forall\vartheta_p\neg\,\langle p,\vartheta\rangle\\
&= \Box\exists\vartheta_p\langle p,\vartheta\rangle
\end{aligned}
$$

The semantics of $\exists\vartheta\,\Box\,\langle p,\vartheta\rangle$ and $\Box\exists\vartheta\langle p,\vartheta\rangle$ will be given in Definition 14. However, we can anticipate that $\exists\vartheta\Box\langle p,\vartheta\rangle \neq \Box\exists\vartheta\langle p,\vartheta\rangle$: as discussed in the introduction of the section, in our approach the interpretation of a predicate can change in the states of a computation. In other words, in σ-formulae the equivalence $\Box_\sigma \equiv \neg_\sigma \Diamond_\sigma \neg_\sigma$ does not hold. On the contrary, the *Barcan formula* $\forall\,\Box \equiv \Box\,\forall$ is a theorem with respect to our semantics. In [53], we show that, exploiting the translation function t, a set of interesting equivalencies can be found also in the case of σ-formulae.

Example 12.

$$
\begin{aligned}
t(p(\mathrm{x}) \wedge_\sigma q(\mathrm{x})) &= \exists\vartheta_x.\,(p(\mathrm{x}) \wedge q(\mathrm{x}))\\
t[p(\mathrm{x}) \wedge_\sigma (p(\mathrm{x}) \rightarrow_\sigma q(\mathrm{x}))] &= \exists\vartheta_x.\ p(\mathrm{x}) \wedge \forall\vartheta_x.\ p(\mathrm{x}) \rightarrow q(\mathrm{x}))
\end{aligned}
$$

This example shows that state $\{p(a), p(b), q(a)\}$ is a model for the first formula, while it is not a model for the second one.

We introduce the notion of *normal form* for the translation of a σ-formula. Normal forms are needed to define the semantics of σ-formulae in an inductive way (see Definition 14). Let A be a σ-formula, we say that $t(A)$ is in *normal form* if for any sub-formula $\langle p, \vartheta\rangle$ of $t(A)$, $p = q\vartheta'$ with q state formula.

For instance, $\exists\vartheta_p\langle p, \vartheta\rangle \vee \exists\vartheta_q\langle q, \vartheta\rangle$ is in normal form, $\exists\vartheta_p[\exists\vartheta'_{q_{\setminus cv}}\langle p \wedge q, \vartheta'\rangle, \vartheta]$ is not. However, since ϑ and ϑ' refer to different sets of variables (this is always the case by Definition 10), we can pass the substitutions through the brackets and obtain the normal form: $\exists\vartheta_p\exists\vartheta'_{q_{\setminus cv}}\langle p \wedge q, \vartheta'\vartheta\rangle$.

Definition 13 (*Models*). A model $\mathcal{M}$ is a pair $\langle S, V\rangle$ with S infinite Noetherian denumerable chain of states (multisets), and V valuation function mapping a state $s \in S$ in the set $\{\langle A, \vartheta\rangle \mid s \models \langle A, \vartheta\rangle\}$

Definition 14 (*Semantics*). Let $\mathcal{M} = \langle S, V\rangle$ be a model; we say that

$$\mathcal{M} \text{ satisfies } p \quad \text{iff} \quad \mathcal{M}, s_0 \|\models t(p) \tag{1}$$

where $\mathcal{M}, s_i \|\models t(p)$ is inductively defined as follows (F denotes a sub-formula). The definition is standard. We provide it, for the sake of completeness:

$\mathcal{M}, s_i \|\models \langle A, \vartheta\rangle$ iff $\langle A, \vartheta\rangle \in V(s_i)$
with A state form., i.e., iff $s_i \models \langle A, \vartheta\rangle$,
$\mathcal{M}, s_i \|\models \exists(\forall)\vartheta F$ iff $\exists(\forall)\vartheta\, \mathcal{M}, s_i \|\models F$
$\mathcal{M}, s_i \|\models \neg F$ iff not $\mathcal{M}, s_i \|\models F$
$\mathcal{M}, s_i \|\models (F_1 \rightarrow F_2)$ iff $\mathcal{M}, s_i \|\models F_1 \rightarrow \mathcal{M}, s_i \|\models F_2$
$\mathcal{M}, s_i \|\models \Box F$ iff $\forall k \geqslant i.\, \mathcal{M}, s_k \|\models F$
$\mathcal{M}, s_i \|\models \Diamond F$ iff $\exists k \geqslant i.\, \mathcal{M}, s_k \|\models F$
$\mathcal{M}, s_i \|\models \bigcirc F$ iff $\mathcal{M}, s_{i+1} \|\models F$
$\mathcal{M}, s_i \|\models F_1 \wedge (\vee) F_2$ iff $\mathcal{M}, s_i \|\models F_1$ and (or) $\mathcal{M}, s_i \|\models F_2$

Example 15. Let $S = \{p(a)\}, \{p(a), p(b), q(a)\}, \{p(b)\}, \{p(b)\}, \{p(b)\}, \ldots$ be a computation, then $\langle S, V\rangle$ satisfies the formulae $\Diamond_\sigma \Box_\sigma p(\mathrm{X})$ and $p(\mathrm{X})\mathrm{UNLESS}_\sigma q(\mathrm{X})$, while it is does not satisfy $\Box_\sigma p(\mathrm{X})$, nor $\Box_\sigma \neg_\sigma (p(\mathrm{X}) + p(\mathrm{Y}))$.

The next proposition states that the semantics of $\sim p(\bar{\mathrm{x}})$, seen as σ-formula, is equivalent to the semantics of $\neg p(\bar{\mathrm{x}})$, where "$\sim$" is the negation symbol we used in state formulae, and "$\neg$" the negation symbol we introduced when extending state formulae with temporal operators (see, respectively, Definitions 1 and 7). This result allows us not to distinguish $\neg$ and $\sim$ in the following.

Proposition 16. $\sim p(\bar{x}) \equiv \neg p(\bar{x})$

Proof. Let $\mathcal{M} = \langle S, V \rangle$ be a model, and s a state in S:

$$
\begin{aligned}
&\mathcal{M}, s \models t(\sim p(\bar{x})) \\
&\Leftrightarrow \mathcal{M}, s \models \exists \vartheta_{p_{\setminus ex}} (\sim p(\bar{x}), \vartheta) \\
&\Leftrightarrow \exists \vartheta_{p_{\setminus ex}}\ \mathcal{M}, s \models (\sim p(\bar{x}), \vartheta) \\
&\Leftrightarrow \exists \vartheta_{p_{\setminus ex}}\ (\sim p(\bar{x}), \vartheta) \in V(s) \\
&\Leftrightarrow \exists \vartheta_{p_{\setminus ex}}\ s \models (\sim p(\bar{x}), \vartheta) \\
&\Leftrightarrow \exists \vartheta_{p_{\setminus ex}}\ s \models_{\mathrm{w}} (\sim p(\bar{x}), \vartheta) \quad \text{with } \vartheta \text{ success substitution for } \sim p(\bar{x}) \text{ and } s \\
&\Leftrightarrow s \models_{\mathrm{w}} (\sim p(\bar{x}), \varepsilon) \\
&\Leftrightarrow \forall \vartheta_p \ \textit{not}\ s \models_{\mathrm{w}} (p(\bar{x}), \vartheta \circ \varepsilon) \\
&\Leftrightarrow \forall \vartheta_p \ \textit{not}\ s \models_{\mathrm{w}} (p(\bar{x}), \vartheta) \quad \text{with } \vartheta \text{ success substitution for } p(\bar{x}) \text{ and } s \\
&\Leftrightarrow \forall \vartheta_p \ \textit{not}\ s \models (p(\bar{x}), \vartheta) \\
&\Leftrightarrow \forall \vartheta_p \ \textit{not}\ (p(\bar{x}), \vartheta) \in V(s) \\
&\Leftrightarrow \forall \vartheta_p \ \textit{not}\ \mathcal{M}, s \models (p(\bar{x}), \vartheta) \\
&\Leftrightarrow \forall \vartheta_p\ \mathcal{M}, s \models \neg (p(\bar{x}), \vartheta) \\
&\Leftrightarrow \mathcal{M}, s \models \forall \vartheta_p \neg (p(\bar{x}), \vartheta) \\
&\Leftrightarrow \mathcal{M}, s \models t(\neg p(\bar{x})) \qquad \square
\end{aligned}
$$

3.3. Non-standard operators

We introduce three new temporal operators. They are a sort of *past* operators and are introduced to lead to more readable and concise specifications, when describing reactive systems as the ones we are taking into consideration. In particular, they allow us to specify conditions related to the appearance (instead of presence) of a tuple in the tuple space. In other words, they allow us to directly relate state changes to state configurations or to further reactions, i.e., to naturally relate transitions to state configurations.

The operators we introduce are: NEEDS, AFTER and $\hookrightarrow$ (CAUSES).[5] The operator NEEDS permits to express a *consistency* condition by constraining the states in which a new property may become true: p NEEDS q means that, if p appears, then q must be true as well. If p is true initially, then q must be true in the initial state too. With p AFTER q we define *necessary* causes: p can become true only if q was true in the previous state. Finally, the property $p \hookrightarrow q$ defines *sufficient* causes: it requires a state satisfying the conclusion q to follow (not necessarily immediately) the state in which the premise p

[5] Warning: from now on we drop label σ, since we use only σ-formulae.

Table 4
Informal definition of temporal operators: predicates over the line refer to the time interval; predicates under the line refer to the time instant; predicates in boldface are those "caused" by the other conditions

$p(X)$ UNLESS $q(X)$ once true, $p(X)$ holds until $q(X)$ becomes true	**p(a)** $p(a)$ $q(a)$
STABLE $p(X)$ once true, $p(X)$ stays true	**p(a)** $p(a)$
$p(X) \mapsto q(X)$ $q(X)$ eventually true after $p(X)$ was true	$p(a)$ **q(a)** $\hat{\bigcirc}$ consecutive states
$p(X)$ NEEDS $q(X)$ $q(X)$ must be true when $p(X)$ becomes true	$\neg p(a)$ $p(a)+$**q(a)** $\hat{\bigcirc}$ consecutive states
$p(X)$ AFTER $q(X)$ $q(X)$ must be true before $p(X)$ becomes true	**q(a)**$+\neg p(a)$ $p(a)$ $\hat{\bigcirc}$ cons. st.
$p(X) \hookrightarrow q(X)$ $q(X)$ eventually true after $p(X)$ became true	$\neg p(a)$ $p(a)$ **q(a)**

becomes true, or to hold sometimes if p is true in the initial state. Formally, let $\hat{\bigcirc}$ indicate the previous state [42], and have the following semantics:

$$\mathcal{M}, s_0 \models t(\hat{\bigcirc} p)$$
$$\mathcal{M}, s_{i+1} \models t(\hat{\bigcirc} p) \quad \text{iff} \quad \mathcal{M}, s_i \models t(p)$$

Then, we define

$$p \text{ NEEDS } q = \text{INIT}(p \rightarrow q) \wedge \Box [\, p \rightarrow (\hat{\bigcirc} p \vee q)\,]$$
$$p \text{ AFTER } q = \Box [\, p \rightarrow \hat{\bigcirc}(p \vee q)\,]$$
$$p \hookrightarrow q = p \text{ NEEDS } (\Diamond q)$$

As an immediate consequence of these definitions, we have the equivalence

$$p \text{ AFTER } q \equiv p \text{ NEEDS } \hat{\bigcirc} q$$

In Table 4 we provide a representation of correct behaviors with respect to formulae including these operators and Unity operators. In the following, we list some useful properties of the new operators (the proofs are in [53]).

3.3.1. Needs

As an example of the use of NEEDS, imagine that somebody (P) wants to buy a software product S, and wants the most recent version V of S. With a natural interpretation of the predicates, this can be expressed as

$$has(P, V, S) \text{ NEEDS } most\_recent(V, S)$$

saying precisely that V is the most recent version of S when P gets it: remember that the left-hand side of NEEDS refers to the state *transition* that makes the operand true.

Actually, a new version of S can be released afterwards: the formula above permits P to continue and use the previous one, if he wants to. Such freedom would not be modeled by

$$has(P,V,S) \rightarrow most\_recent(V,S)$$

which forces P to keep updating the version he has. On the other side, the best we can do with UNLESS is

$$(\neg has(P,V,S)) \text{ UNLESS } most\_recent(V,S)$$

which, besides being definitely less readable, would allow P to get a version V of S (immediately) before its release, thus not describing precisely what we want to specify.

Operator NEEDS is reflexive (N1, Table 5), distributes over the conjunction (N2), enjoys a closure property with respect to implication (N3), and entails two obligations, let p NEEDS q hold: if q is never satisfied, then p can never become true (N4), and if $\neg q$ NEEDS $\neg p$ holds as well, then q must be true whenever p is true (N5). Finally, rule N6 states a weak transitivity property.

3.3.2. *After*

To exemplify the use of AFTER, we consider flight reservations. A safety condition that we might want to describe is that a customer P gets a reservation on a flight F only if he made a request and if there were some seats free before the reservation is taken, i.e.

$$reserved(P,F) \text{ AFTER } (seats\_available(F) + request(P,F))$$

Again, the use of operators $\rightarrow$ and $\hat{\bigcirc}$, as in

$$reserved(P,F) \rightarrow \hat{\bigcirc}\,(seats\_available(F) + request(P,F))$$

results in a too restrictive condition, and UNLESS forces the introduction of a negation, making the formula less explicit:

$$(\neg reserved(P,F)) \text{ UNLESS } (seats\_available(F) + request(P,F))$$

The properties of AFTER listed in Table 5 are similar to those of NEEDS : it is anti-reflexive (A1), distributes over the conjunction (A2), it shows a transitive closure (A3), and entails an obligation (A4). Finally, A5 relates AFTER and NEEDS.

3.3.3. *Causes*

To describe CAUSES, we compare it with the stronger $\mapsto$. First of all, the following equivalence holds:

$$p \mapsto q \equiv p \rightsquigarrow q \wedge [(p \wedge q) \text{ UNLESS } (\neg p \vee \Diamond q)]$$

In other words, to satisfy a property expressed with $\mapsto$ we cannot falsify the conclusions without also falsifying the premises or ensuring that the conclusion will be verified once

Table 5
Properties of the new operators. In **N2** and **A2** we require q and r to share only variables that appear also in p

Properties of NEEDS		
p NEEDS p	(reflexivity)	**N1**
$(p \text{ NEEDS } q) \wedge (p \text{ NEEDS } r) \equiv p \text{ NEEDS } (q \wedge r)$	($\wedge$ equivalence)	**N2**
$\dfrac{p \text{ NEEDS } q \qquad \Box(q \rightarrow r)}{p \text{ NEEDS } r}$	(NEEDS– $\rightarrow$)	**N3**
$\dfrac{p \text{ NEEDS } q}{\Box(\neg q) \rightarrow \Box(\neg p)}$	(NEEDS – obligation 1)	**N4**
$(p \text{ NEEDS } q) \wedge (\neg q \text{ NEEDS } \neg p) \equiv \Box(p \rightarrow q)$	(NEEDS – obligation 2)	**N5**
$\dfrac{p \text{ NEEDS } q \qquad q \text{ NEEDS } r \qquad r \text{ UNLESS } \neg q}{p \text{ NEEDS } r}$	(weak – transitivity)	**N6**
Properties of AFTER		
p AFTER $\neg p$	(anti-reflexivity)	**A1**
$(p \text{ AFTER } q) \wedge (p \text{ AFTER } r) \equiv p \text{ AFTER } (q \wedge r)$	($\wedge$ equivalence)	**A2**
$\dfrac{p \text{ AFTER } q \qquad \Box(q \rightarrow r)}{p \text{ AFTER } r}$	(AFTER– $\rightarrow$)	**A3**
$\dfrac{p \text{ AFTER } q \qquad \text{INIT}(\neg p)}{\Box(\neg q) \rightarrow \Box(\neg p)}$	(AFTER- obligation)	**A4**
$p \text{ AFTER } q \equiv p \text{ NEEDS } \hat{\bigcirc} q$		**A5**
Properties of CAUSES		
$p \hookrightarrow p$	(reflexivity)	**C1**
$\dfrac{p \hookrightarrow q}{\Box(\neg q) \rightarrow \Box(\neg p)}$	($\hookrightarrow$ –obligation)	**C2**
$\dfrac{p \hookrightarrow q \qquad \Box(q \rightarrow r)}{p \hookrightarrow r}$	($\hookrightarrow$ – $\rightarrow$)	**C3**
$p \mapsto q \equiv [(p \wedge q) \text{ UNLESS } (\neg p \vee \Diamond q)] \wedge p \hookrightarrow q$		**C4**
$\dfrac{p \mapsto q}{p \hookrightarrow q}$		**C5**
$\dfrac{p \hookrightarrow q \qquad q \mapsto r}{p \hookrightarrow r}$		**C6**

more. This is why the use of $\hookrightarrow$ is very useful in the earliest stages of a refinement, when it is more convenient to say something like: "if p appears, then ..." deferring the choices related to p staying true or not. For instance, let us consider

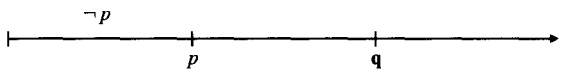
$$request(R) \hookrightarrow answer(R,A)$$

Its premise says that there is a request R. The consequence states that there is an answer A to R. The formula itself states that, following the state in which the premise

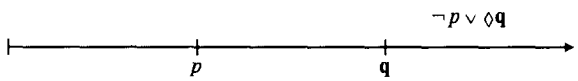
becomes true, the process will reach a state, in which the consequence holds: nothing else is implied on future evolutions.

Note that, in tuple spaces, the natural way to falsify a condition is to consume a tuple: for instance, once the answer has been taken into consideration, it is natural to consume it. With the formula above, the refiner is free to follow this line. If, on the contrary, we use $\mapsto$, we force the refiner to consume also some piece of the premises, when he wants to consume the consequence.

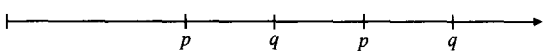
To sum up, we show models of $p \mapsto q$ and of $p \hookrightarrow q$. The property $p \hookrightarrow q$ requires a state satisfying the conclusion (q) to follow the states in which the premise (p) *becomes* true. For instance, the following computation[6] is a model for $p \hookrightarrow q$:



i.e. each state in which p becomes true is followed by a state in which q is true. There are no other constraints: for instance, p can stay true even after the state in which q is true, without requiring q to stay or become true again. Instead, a computation which is a model for $p \mapsto q$ has to satisfy more constraints:



A state in which p is true is followed by one in which q is true: to let q undetermined after it appeared, we have to negate p. If p continues to be true after q became true, then further states with q true are required:



The properties of $\hookrightarrow$ are also listed in Table 5: it is reflexive (C1), entails an obligation (C2), and shows a transitive closure (C3). Some properties relate $\hookrightarrow$ and $\mapsto$: the first operator is weaker than the second one (C4 and C5); they can be combined to derive a transitivity property (C6).

4. Semantics

We define both an axiomatic and an operational semantics for the prototype language defined in Section 2. The axiomatic semantics is in the weakest precondition style, and the transition system defining the operational semantics is based on the new notion of enabling precondition, which exploits the angelic choice in non-deterministic selection.

[6] We are using the same conventions introduced in the caption of Table 4.

We relate operational and weakest precondition semantics so that the properties derived using the axiomatic semantics are correct and complete with respect to the operational behavior. This result overcomes the problems related to the basic statement, which is a non-deterministic rule rather than a simple deterministic assignment.

4.1. Motivations

In deterministic sequential languages the operational semantics of a program, given an initial state, is the singleton set containing the only possible computation. It is not so difficult to prove that a program satisfies the specification, i.e. to prove assertions of the kind $\{p\}$ *program* $\{q\}$. The calculus is usually based on a weakest precondition semantics. Its correctness is ensured by the following result. Let $T(S)$ be the transition system defining the operational semantics of an atomic statement S:

$$\langle s, s' \rangle \in T(S) \text{ iff } \forall q\, [s' \models q \leftrightarrow s \models wp(S, q)] \tag{2}$$

Then, the sequential combination $S_1; S_2$ corresponds to $wp(S_1, wp(S_2, q))$ and so on [8].

In most concurrent languages (e.g. Action Systems [10], Unity [23]) the operational semantics is a (possibly infinite) set of computations. All the computations start from the same initial state s_0. Hence, we can think of the set of computations as the set of branches of a tree T rooted in s_0. Let s be a node of the tree, and $s_1, \ldots, s_n$ the sons of s. A transition from s to s_i is defined by one of the (deterministic) statements (S_i) that can be executed in s. In other words, there is non-determinism in statement selection, while after the commitment to a particular statement, the behavior is deterministic. For deterministic statements result (2) relating operational and axiomatic semantics still holds. Verification calculi for these languages are based on it. Their novelty is the way of combining assertions of the kind $\{p\}$ *statement* $\{q\}$ to prove that programs satisfy specifications even in case of non-deterministic choices (see, for instance [23]).

In tuple space languages (2) does not hold. Statements are rules and behave non-deterministically. A rule, when executed in a state s can lead to different states, depending upon which atoms in s are selected to be read and consumed by the rule guard.

Example 17. Consider the rule

$$R: \textbf{cons } d(\mathrm{X}) \textbf{ ask } \mathrm{X} > 0 \mid \textbf{body } \mathrm{Y} \textit{ is } \mathrm{X} - 1. \textbf{ tell } d(\mathrm{Y})$$

The intended meaning is that, given $s = \{d(2), d(4)\}$, transitions:

$$s \rightarrow \{d(1), d(4)\} \quad \text{and} \quad s \rightarrow \{d(2), d(3)\}$$

are allowed, and therefore $\{d(1), d(4)\} \in T(R)(s)$. On the other side, given $q = d(1) + d(4)$, $s \not\models wp(R, q)$, since from s we may end in $\{d(2), d(3)\}$ that does not satisfy q.

We found, for tuple space languages, equivalence results corresponding to (2), namely Theorems 49 and 50.

Non-determinism has been studied by Back and von Wright in [11]. We summarize below the results of interest. In Section 4.3 we define the weakest precondition of a rule, in Section 4.4, we introduce the notion of enabling precondition, and we use it to define an operational semantics in Section 4.5.

According to Back and von Wright, a *demonic strict* assignment has syntax: $\wedge x := x'.c$, where x, x' are variables, and c is a predicate on x'. The statement $x := x'.c$ assigns to x any value x' satisfying c. As an example, $x := x'.\{x' \in \mathbf{Z}\}$ assigns to x any integer. The weakest precondition of a strict demonic assignment statement is defined as follows.

$$wp(\wedge x := x'.c,\ q) = \exists x'.c \ \wedge\ \forall x'.(c \rightarrow q[x'/x])$$

The weakest precondition ensures that a state satisfying the postcondition is reached even with a *demonic* choice of the value x' satisfying c, i.e. one that tries to abort the statement. For instance,

$$wp(\wedge x := x'.\{x' \in \mathbf{N}\},\ x \geqslant 0) = true$$
$$wp(\wedge x := x'.\{x' \in \mathbf{N}\},\ x \neq 7) = false$$

In the case of *angelic strict* assignment we have

$$wp(\vee x := x'.c, q) = \exists x'.(c \ \wedge\ q[x'/x])$$

If the weakest precondition is satisfied, then a state satisfying the postcondition is reached with an angelic choice of the value x' satisfying c, i.e. one that tries to execute the statement. For instance,

$$wp(\vee x := x'.\{x' \in \mathbf{N}\},\ x \geqslant 0) = true$$
$$wp(\vee x := x'.\{x' \in \mathbf{N}\},\ x = 4) = true$$

The only sensible choice for the weakest precondition of a rule is the demonic behavior since it ensures that a state satisfying the postcondition is reached from the state satisfying the precondition, independently from the choices in the rule. The angelic behavior will be introduced to define the operational semantics in Section 4.4.

Before going into the technicalities, and to better understand them, we need to compare our work with Swarm [28]. Unity influences the proof logic for Swarm as well. Like in Unity, the building blocks of the proofs in Swarm are Hoare triples. However, Swarm is a tuple space language, and the basic statement is read-write statement on the tuple space rather than an assignment. Consequently, the usual proof rule to derive assertions of the kind $\{p\}\, R\, \{q\}$ cannot be used. In Swarm, a transaction relation predicate *step* is defined: $step(t, R, t')$ means that rule R is in tuple space t, and its execution can transform t in t'. The meaning of the assertion $\{p\}\ R\ \{q\}$ is then supplied saying

$$\{p\}\ R\ \{q\} \ \equiv\ \forall t, t'.\ step(t, R, t') :\ \ p(t) \Rightarrow q(t')$$

However, a constructive derivation for Hoare triples in the case of rules was missing. The definition of a formal method to prove $\{p\}\,R\,\{q\}$, allowing this gap to be closed, is one of the results of our work, the difficulty being caused by the non-deterministic features of the rule discussed above. Moreover, Hoare triples are too strong when proving liveness properties. When refining liveness, we do not need to require that

$$\forall t, t'.\ step(t, R, t') :\ p(t) \Rightarrow q(t').$$

In fact, it is sufficient that a fair selection is guaranteed and to show that

$$\forall t.\ p(t) :\ \exists t'.\ step(t, R, t') \ \wedge\ q(t').$$

We introduce the notion of rule *enabling precondition*, capturing this condition, and provide a formal derivation for it. When we say that $p \rightarrow ep(R, q)$, we say that rule R can fire in every state s satisfying p and *possibly* lead to state s' satisfying q. Note that (1) says that *wp* and *ep* coincide in the case of deterministic statements, while in the non-deterministic ones *ep* is implied by *wp* (see Proposition 44).

4.2. Preliminaries

We need some preliminaries. The reader can skip this section now, and use it as a reference while reading the next ones. We use here a different notation for the state formulae introduced in Section 3.1. This notation is less readable, but suits better the topic of this section. Here and in the following, $\bar{\mathrm{x}}$ indicates an array of variables.

A state formula is here a formula built over:

(1) Conditions $p(\bar{\mathrm{x}})^n$ with n natural number, constraining the multiplicity of atom $p(\bar{\mathrm{x}})$ in the state to be at least n. We omit exponent n when it equals 1.
(2) Conditions $\mathrm{X} \neq \mathrm{Y}$, $\mathrm{X} = \mathrm{Y}$, $\mathrm{X}\ is\ \mathrm{Y} + 3$ ... constraining variables X, Y appearing in the other condition (primitives inherited from Prolog.)
(3) Classical connectives and quantifiers of 1^{st} order logic: $\wedge$, $\vee$, $\rightarrow$, $\neg$, $\forall$, $\exists$.

As an example, $p(a)^3 \wedge \neg\, p(a)^5$ means that we want the occurrences of $p(a)$ to be 3 or 4.

Definition 18. We introduce an algebra for the exponents. It permits also to translate in the new notation state formulae including the non-idempotent conjunction $+$. Let A, B be atoms, and F a formula

$$\mathrm{A}^n + \mathrm{B}^m \equiv [\mathrm{A} \neq \mathrm{B} \wedge \mathrm{A}^n \wedge \mathrm{B}^m] \quad \vee \quad [\mathrm{A} = \mathrm{B} \wedge \mathrm{A}^{n+m}]$$

$$\neg\mathrm{A}^n + \mathrm{F} \equiv \neg\mathrm{A}^n \wedge \mathrm{F}$$

$$\mathrm{A}^n \wedge \mathrm{A}^m \equiv \mathrm{A}^{max(n,m)}$$

$$\mathrm{A}^n \vee \mathrm{A}^m \equiv \mathrm{A}^{min(n,m)}$$

$$\mathrm{A}^0 \equiv true$$

Example 19.

$$\begin{aligned}
\neg \mathrm{A}^n + \neg \mathrm{A}^m &\equiv \neg \mathrm{A}^n \wedge \neg \mathrm{A}^m \\
&\equiv \neg(\mathrm{A}^n \vee \mathrm{A}^m) \\
&\equiv \neg \mathrm{A}^{min(n,m)}
\end{aligned}$$

$$p(\mathrm{X})^2 \wedge p(\mathrm{Y}) \equiv [\mathrm{X} = \mathrm{Y} \wedge p(\mathrm{X})^2] \vee [\mathrm{X} \neq \mathrm{Y} \wedge p(\mathrm{X})^2 \wedge p(\mathrm{Y})]$$

$$p(\mathrm{X})^2 + p(\mathrm{Y}) \equiv [\mathrm{X} = \mathrm{Y} \wedge p(\mathrm{X})^3] \vee [\mathrm{X} \neq \mathrm{Y} \wedge p(\mathrm{X})^2 \wedge p(\mathrm{Y})]$$

The semantics of the state formulae is based on Definition 14. In particular, we require all the variables occurring in a formula p to be bound in order to say $s \models p$. The operators defined here, as well as the *wp* derivation, are syntax driven, and can be easily automated.

The following transformations are based on the fact that the multiplicity of an atom $a(\bar{\mathrm{X}})$ appearing in the postcondition is affected by a tell (consume) operation **tell** $t(\bar{\mathrm{B}})$ (**cons** $t(\bar{\mathrm{B}})$) only when $a = t$ and the arguments $\bar{\mathrm{X}}$ and $\bar{\mathrm{B}}$ coincide.

Definition 20 (*Atomic unifying transformation*). We define the atomic unifying transformation of a pair of atoms $a(\bar{\mathrm{X}})$, $t(\bar{\mathrm{B}})$ as follows. We require variable names in $a(\bar{\mathrm{X}})$ and $t(\bar{\mathrm{B}})$ to be distinguished:

$$aut(A(\bar{\mathrm{X}})^n, t(\bar{\mathrm{B}})) = \begin{cases} a(\bar{\mathrm{X}})^n & \text{if } a \neq t, \\ [t(\bar{\mathrm{B}})^n \wedge \bar{\mathrm{X}} = \bar{\mathrm{B}}] \vee [t(\bar{\mathrm{X}})^n \wedge \bar{\mathrm{X}} \neq \bar{\mathrm{B}}] & \text{if } a = t. \end{cases}$$

Definition 21 (*Unifying transformation*). Let q be a state formula, $t(\bar{\mathrm{B}})$ an atom:

$$ut(q, t(\bar{\mathrm{B}})) = q[a(\bar{\mathrm{X}})^n / aut(a(\bar{\mathrm{X}})^n, t(\bar{\mathrm{B}}))] \text{ for every atom } a(\bar{\mathrm{X}}) \text{ in } q.$$

We translate rule guards into state formulae. Without loss of generality, we consider rule guards with the structure ASK CONSUME (i.e. all the ask conditions precede the consume conditions).

Definition 22 (*Guards to state formulae*).

$$g2f(\text{ASK CONSUME}) = a2f(\text{ASK}) \wedge c2f(\text{CONSUME})$$

$$a2f(\text{ASK}, \textbf{ask } A) = a2f(\text{ASK}) \wedge A$$

$$c2f(\text{CONSUME}, \textbf{cons } A) = c2f(\text{CONSUME}) + A$$

Remark 23. The formulae produced by $g2f$ have to be normalized according to Definition 2 (like in the fourth equation of the next example), and to Definition 18.

Example 24 (*Guards to state formulae*).

$$g2f(\textbf{ask } p(\mathrm{X})\ \textbf{ask } p(\mathrm{Y})\ \textbf{ask } \mathrm{X} > 0) = p(\mathrm{X}) \wedge p(\mathrm{Y}) \wedge \mathrm{X} > 0$$

$$g2f(\textbf{cons } p(\bar{\mathrm{X}})\ \textbf{cons } p(\bar{\mathrm{Y}})) = [p(\bar{\mathrm{X}})^2 \wedge \bar{\mathrm{X}} = \bar{\mathrm{Y}}] \vee [p(\bar{\mathrm{X}}) \wedge p(\bar{\mathrm{Y}}) \wedge \bar{\mathrm{X}} \neq \bar{\mathrm{Y}}]$$

$$g2f(\textbf{cons } p(\bar{\mathrm{Y}})\ \textbf{cons } p(\bar{\mathrm{Y}})\ \textbf{ask } p(\bar{\mathrm{X}})\ \textbf{ask } \neg q(\bar{\mathrm{Y}})) = p(\bar{\mathrm{Y}})^2 \wedge \neg q(\bar{\mathrm{Y}}) \wedge p(\bar{\mathrm{X}})$$

$$g2f(\textbf{ask } p(\mathrm{X})\ \textbf{ask } \mathrm{X} > \mathrm{Y}\ \textbf{ask } \neg p(\mathrm{Y})) = p(\mathrm{X}) \wedge \neg(\mathrm{X} > \mathrm{Y} \wedge p(\mathrm{Y}))$$

4.3. Weakest precondition semantics

We define in this section the weakest precondition semantics for TuSpReL. In the next section we complete the axiomatic semantics by introducing the enabling precondition semantics, exploit it to define an operational semantics, and relate operational and axiomatic semantics.

Definition 25 (*Rule weakest precondition*). Let R be the rule: GUARD | BODY TELL_1; TELL_2 with ASK and CONSUME lists of ask and consume conditions of the guard, respectively, and q be a state property. Equations are commented below:

$$wp(R, q) = \tag{3}$$

$$wp(\mathrm{GUARD}, wp(\mathrm{CONSUME}, wp(\mathrm{BODY}, \{wp(\mathrm{TELL}_1, q), wp(\mathrm{TELL}_2, q)\})))$$

$$wp(\textbf{tell } t(\bar{\mathrm{B}}), q) = ut(q, t(\bar{\mathrm{B}}))[t(\bar{\mathrm{B}})^n / t(\bar{\mathrm{B}})^{n-1}] \tag{4}$$

$$wp(\textbf{body } t(\bar{\mathrm{B}}), \{q_1, q_2\}) = [t(\bar{\mathrm{B}}) \wedge q_1] \vee [\neg t(\bar{\mathrm{B}}) \wedge q_2] \tag{5}$$

$$wp(\textbf{cons } t(\bar{\mathrm{B}}), q) = ut(q, t(\bar{\mathrm{B}}))[t(\bar{\mathrm{B}})^n / t(\bar{\mathrm{B}})^{n+1}] \tag{6}$$

$$wp(\mathrm{GUARD}, q) = g2f(\mathrm{GUARD}) \wedge [g2f(\mathrm{GUARD}) \rightarrow q] \tag{7}$$

(3) *Rule weakest precondition*: rule R and property q share no variable (always possible by renaming). We recall that postconditions TELL_1 and TELL_2 apply in case of successful or unsuccessful evaluation of the body, respectively.

(4) *Tell weakest precondition:* **tell** $t(\bar{\mathrm{B}})$ is an atomic tell condition. Unifying transformation $ut(q, t(\bar{\mathrm{B}}))$ is defined in Section 4.2. As an example:

$$wp(\textbf{tell } d(\mathrm{Y}), d(\mathrm{Z})^2) = [d(\mathrm{Z}) \wedge \mathrm{Y} = \mathrm{Z}] \vee d(\mathrm{Z})^2$$

In fact, either $\mathrm{Y} = \mathrm{Z}$ or $\mathrm{Y} \neq \mathrm{Z}$: in the first case only one occurrence of $d(\mathrm{Z})$ is needed before the tell operation (the other one is written); in the second case two occurrences of $d(\mathrm{Z})$ are needed before the tell. As a further example,

$$wp(\textbf{tell } d(\mathrm{Y}), \neg d(\mathrm{Z})) = \mathrm{Y} \neq \mathrm{Z} \wedge \neg d(\mathrm{Z})$$

Then, the weakest precondition of the whole TELL is defined as

$$wp(\mathrm{TELL}\ \textbf{tell } t(\bar{\mathrm{B}}), q) = wp(\mathrm{TELL}, wp(\textbf{tell } t(\bar{\mathrm{B}}), q))$$

The result is independent of the order of the tell operations.

(5) *Body weakest precondition*: **body** $t(\bar{B})$ is the body. The two postconditions refer to successful evaluation of the body, or failure, respectively. As an example,

$$wp(\textbf{body } p(Y,Z), \{d(Z)^2, a\}) = [p(Y,Z) \wedge d(Z)^2] \vee [\neg p(Y,Z) \wedge a]$$

If Z is the first computed answer substitution for $p(Y,Z)$, then $d(Z)^2$ has to hold before body evaluation, otherwise, if no computed answer substitution for the body exists, i.e., the body fails (negation as failure), then a is the needed precondition. Note that we are considering that the BODY, being defined by a Prolog program, is deterministic: BODY evaluation returns the first computed substitution, if any.

(6) *Consume weakest precondition*: **cons** $t(\bar{B})$ is an atomic consume condition. As for tell: $ut(q, t(\bar{B}))$ is defined in Section 4.2. As an example,

$$wp(\textbf{cons } d(Y), d(Z)^2) = [d(Z)^2 \wedge_{Y \neq Z}] \vee d(Z)^3$$

in fact, if $Y \neq Z$, then only two occurrences of $d(Z)$ are needed before the tell operation, while if $Y = Z$, i.e. an occurrence of $d(Z)$ is consumed, then three occurrences of $d(Z)$ are needed before the consume.

Then, the weakest precondition of the whole CONSUME is defined as

$$wp(\text{CONSUME } \textbf{cons } t(\bar{B}), q) = wp(\text{CONSUME}, wp(\textbf{cons } t(\bar{B}), q))$$

The result is independent from the order in this case too.

(7) *Guard weakest precondition*: $g2f(\text{GUARD})$ is the translation, in terms of state formulae, of the constraints expressed by the guard, according to Definition 22. As an example, let rule guard G be **cons** $d(Y)$, **ask** $Y > 0$, then

$$wp(G, b(Y,Z)^2) = [d(Y) \wedge Y > 0] \wedge [(d(Y) \wedge Y > 0) \rightarrow b(Y,Z)^2]$$

In other words, there must be an atom unifying with the rule guard and for every unifying instance of Y, two occurrences of $b(Y,Z)$ are needed as precondition. Observe that the weakest precondition of guard G above cannot be simplified since $p \wedge (p \rightarrow q) \not\equiv p \wedge q$, as shown in Example 12.

Remark 26. The weakest precondition of a guard is based on the weakest precondition of the demonic strict assignment. In fact, if we apply function t of Definition 10, we have

$$t(wp(\text{GUARD}, q)) = \exists \bar{X}. \forall \bar{N}. g2f(\text{GUARD}) \wedge \forall \bar{X}. [\forall \bar{N}. g2f(\text{GUARD}) \rightarrow t(q)]$$

where $\bar{X}$ are the variables occurring positively in the guard, $\bar{N}$ those occurring only negatively. Formally, $\bar{X} \equiv pos(\text{GUARD})$, $\bar{N} \equiv vars(\text{GUARD}) \setminus pos(\text{GUARD})$ (see Definition 2).

Table 6
Derivation of a weakest precondition

$$wp(R, d(\mathrm{Z}))$$
$$= d(\mathrm{X}) + \mathrm{X} > 0 \quad \wedge$$
$$[d(\mathrm{X}) + \mathrm{X} > 0] \rightarrow \{[\mathrm{X} = \mathrm{Z} + 1] \vee [d(\mathrm{Z}) + d(\mathrm{Z})] \vee [d(\mathrm{Z}) + \mathrm{Z} \leqslant 0]\}$$

Derivation:

$$d(\mathrm{Z})$$
tell $d(\mathrm{Y})$
$$(d(\mathrm{Y})^0 \wedge \mathrm{Z} = \mathrm{Y}) \vee (d(\mathrm{Z}) \wedge \mathrm{Y} \neq \mathrm{Z}) \equiv \mathrm{Z} = \mathrm{Y} \vee d(\mathrm{Z})$$
body Y *is* $\mathrm{X} - 1$
$$\mathrm{Y} \textit{ is } \mathrm{X} - 1 \wedge [\mathrm{Y} = \mathrm{Z} \vee d(\mathrm{Z})] \equiv \mathrm{Z} = \mathrm{X} - 1 \vee d(\mathrm{Z})$$
cons $d(\mathrm{X})$
$$\mathrm{Z} = \mathrm{X} - 1 \vee (d(\mathrm{Z}) \wedge \mathrm{X} \neq \mathrm{Z}) \vee (d(\mathrm{X})^2 \wedge \mathrm{X} = \mathrm{Z})$$

$$g2f(\textbf{cons } d(\mathrm{X}) \textbf{ ask } \mathrm{X} > 0) = d(\mathrm{X}) \wedge \mathrm{X} > 0$$

$$wp(R, d(\mathrm{Z}))$$
$$= [d(\mathrm{X}) \wedge \mathrm{X} > 0] \wedge$$
$$(d(\mathrm{X}) \wedge \mathrm{X} > 0) \rightarrow \{[\mathrm{X} - 1 = \mathrm{Z}] \vee [d(\mathrm{X})^2 \wedge \mathrm{X} = \mathrm{Z}] \vee [d(\mathrm{Z}) \wedge \mathrm{X} \neq \mathrm{Z}]\}$$
$$= [d(\mathrm{X}) \wedge \mathrm{X} > 0] \wedge$$
$$(d(\mathrm{X}) \wedge \mathrm{X} > 0) \rightarrow \{[\mathrm{X} = \mathrm{Z} + 1] \vee d(\mathrm{Z})^2 \vee [d(\mathrm{Z}) \wedge \mathrm{Z} \leqslant 0]\}$$

In words, a state s, enabling rule R and guaranteeing that the execution of R leads to a state containing $d(\mathrm{Z})$, must both contain an atom satisfying the guard and satisfy at least one of the following conditions:
(1) there are only atoms $d(\mathrm{Z} + 1)$ with predicate d and argument > 0 (one of them is surely taken leading to $d(\mathrm{Z})$);
(2) there are two atoms $d(\mathrm{Z})$ (even if one is taken, one is left);
(3) there are some (one at least) occurrences of $d(\mathrm{Z})$, but $\mathrm{Z} \leqslant 0$, so they stay there.
Let us finally observe that, according to Definition 18, the derived weakest precondition of $d(\mathrm{Z})$ coincides with the one given on the top of the table.

In Table 6 we derive the weakest precondition of the rule of Example 17 to end up in a state satisfying $d(z)$.

Special cases. It is convenient to consider two special cases.

(1) A rule can have a non-empty body and only one tell, i.e. have the form GUARD | BODY TELL (this is actually the case of the rule in Table 6). The meaning is that the body cannot fail. Let the body be $t(\bar{\mathrm{B}})$, then

$$wp(\textbf{body } t(\bar{\mathrm{B}}), \{q', \mathit{false}\}) = (t(\bar{\mathrm{B}}) \wedge q') \vee (\neg t(\bar{\mathrm{B}}) \wedge \mathit{false}) = t(\bar{\mathrm{B}}) \wedge q'$$

and $wp(R, q)$ reduces to

$$wp(R, q) = wp(\text{GUARD}, wp(\text{CONSUME}, t(\bar{\text{B}}) \wedge wp(\text{TELL}, q)))$$

(2) In the second case R has the form GUARD | TELL, i.e. it has an empty body, or, equivalently the body is *true.* and thus omitted. In this case the weakest precondition of the body reduces to the identity function, and

$$wp(R, q) = wp(\text{GUARD}, wp(\text{CONSUME}, wp(\text{TELL}, q)))$$

Example 27. Another interesting example is the weakest precondition of a rule R to end up in a state satisfying *true* : it reduces to require that the rule guard is satisfied. In fact, it is easy to check that

$$\begin{aligned} wp(\text{TELL}, true) &\equiv true \\ wp(\text{BODY}, \{true, true\}) &\equiv true \\ wp(\text{CONSUME}, true) &\equiv true \\ wp(\text{GUARD}, true) &\equiv g2f(\text{GUARD}) \end{aligned}$$

On the other side, the weakest precondition of a rule to end up in a state satisfying *false* is *false* (see Proposition 29).

4.3.1. Properties

The following results state some properties of the weakest precondition. All the proofs are in Appendix A.

Proposition 28. *Let p and q be state formulae and R a rule, then*

$$wp(R, p \wedge q) \equiv wp(R, p) \wedge wp(R, q)$$

Proposition 29. *Let R be a rule. Then*

$$wp(R, false) \equiv false$$

Corollary 30. *Let p and q be state formulae and R a rule, then*

$$\frac{p \rightarrow q}{wp(R, p) \rightarrow wp(R, q)}$$

Corollary 31. *Let p be a state formula and R a rule, then*

$$wp(R, \neg p) \rightarrow \neg wp(R, p)$$

Corollary 32. *Let p and q be state formulae and R a rule, then*

$$wp(R, p) \vee wp(R, q) \rightarrow wp(R, p \vee q)$$

4.4. Towards an operational semantics: enabling preconditions

We need to define an operational semantics for our language, to describe the allowed computations of a system. We introduce here the (new) notion of enabling precondition (*ep*), and show how to derive the enabling precondition of a rule. In Section 4.6 we relate weakest and enabling preconditions. The definition of the operational semantics, based on the enabling precondition, will be given in Section 4.5, together with a discussion on our not standard approach.

As discussed in Section 4.1, we cannot define the transitions of a rule R as the pairs of states

$$\langle s, s' \rangle \text{ s.t. } \forall q\, [s' \Vdash q \leftrightarrow s \Vdash wp(S, q)]$$

because of the non-determinism of the rules. The enabling precondition actually captures the operational behavior of a rule and overcomes the problem discussed in Section 4.1. For instance, consider rule R of Example 17, and take $q = d(1) \wedge d(4)$. We have

$$\{d(2), d(4)\} \not\Vdash wp(R, q)$$

while we will find that

$$\{d(2), d(4)\} \Vdash ep(R, q)$$

4.4.1. Enabling precondition semantics

We make use here of some notions introduced in Sections 4.2 and 4.3, in particular we refer to Definition 25 and to Definition 22.

Definition 33 (*Rule enabling precondition*). Let q be a state property, R a rule: GUARD | BODY TELL_1; TELL_2 with ASK and CONSUME lists of ask and consume conditions of the guard, respectively. We also require that rule R and property q share no variable.

$$ep(R, q) = ep(\text{GUARD}, wp(\text{CONSUME}, wp(\text{BODY}, \{wp(\text{TELL}_1, q), wp(\text{TELL}_2, q)\})))$$

$$ep(\text{GUARD}, q) = g2f(\text{GUARD}) \wedge q$$

Remark 34. The difference among enabling precondition and weakest precondition of a rule relies on the treatment of the guard. The conditions derived in the two cases are different, as shown in Example 12. The discussion of a complex example is in Remark 38.

Special cases. Definition 33 is simplified if rule R shapes GUARD | BODY TELL or GUARD | TELL. In these cases $ep(R, q)$ reduces, respectively, to

$$ep(\text{GUARD}, wp(\text{CONSUME}, wp(\text{BODY}, wp(\text{TELL}, q))))$$

Table 7
Derivation of an enabling precondition

let R : **cons** $n(\mathrm{Z})$ **ask** $n(\mathrm{Y})$ **ask** $\mathrm{Z}\ mod\ \mathrm{Y} = 0$ *ask* $\mathrm{Z} > \mathrm{Y}$ |

$$ep(R, \neg n(\mathrm{X})) = [n(\mathrm{X}) + n(\mathrm{Y}) + \neg(n(\mathrm{X}) + n(\mathrm{X})) \wedge \mathrm{X}\ mod\ \mathrm{Y} = 0 \wedge \mathrm{X} > \mathrm{Y}] \vee$$
$$[n(\mathrm{Z}) + n(\mathrm{Y}) + \neg n(\mathrm{X}) \wedge \mathrm{Z}\ mod\ \mathrm{Y} = 0 \wedge \mathrm{Z} > \mathrm{Y}]$$

Derivation:

$\neg n(\mathrm{X})$

cons $n(\mathrm{Z})$

$[\mathrm{Z} = \mathrm{X} \wedge \neg n(\mathrm{X})^2] \vee [\mathrm{X} \neq \mathrm{Z} \wedge \neg n(\mathrm{X})]$

$g2f($**cons** $n(\mathrm{Z})$ **ask** $n(\mathrm{Y})$ **ask** $\mathrm{Z}\ mod\ \mathrm{Y} = 0$ **ask** $\mathrm{Z} > \mathrm{Y})$

$$= n(\mathrm{Z}) \wedge n(\mathrm{Y}) \wedge \mathrm{Z}\ mod\ \mathrm{Y} = 0 \wedge \mathrm{Z} > \mathrm{Y}$$

$$ep(R, \neg n(\mathrm{X})) = n(\mathrm{Z}) \wedge n(\mathrm{Y}) \wedge \mathrm{Z}\ mod\ \mathrm{Y} = 0 \wedge \mathrm{Z} > \mathrm{Y} \wedge$$
$$[(\mathrm{Z} = \mathrm{X} \wedge \neg n(\mathrm{X})^2) \vee (\mathrm{X} \neq \mathrm{Z} \wedge \neg n(\mathrm{X}))]$$
$$= [n(\mathrm{X}) \wedge n(\mathrm{Y}) \wedge \neg n(\mathrm{X})^2 \wedge \mathrm{X}\ mod\ \mathrm{Y} = 0 \wedge \mathrm{X} > \mathrm{Y}] \vee$$
$$[n(\mathrm{Z}) \wedge n(\mathrm{Y}) \wedge \neg n(\mathrm{X}) \wedge \mathrm{Z}\ mod\ \mathrm{Y} = 0 \wedge \mathrm{Z} > \mathrm{Y}]$$

and

$$ep(\text{GUARD}, wp(\text{CONSUME}, wp(\text{TELL}, q)))$$

Remark 35. We observe that the enabling precondition of a guard is based on the weakest precondition of the angelic strict assignment. In fact, if we apply function t of Definition 10, we have

$$t(ep(\text{GUARD}, q)) = t(g2f(\text{GUARD}) \wedge q) = \exists \bar{\mathrm{X}}.[\forall \bar{\mathrm{N}}.g2f(\text{GUARD}) \wedge t(q)]$$

where $\bar{\mathrm{X}}$ are the variables occurring positively in the guard, $\bar{\mathrm{N}}$ those occurring only negatively.

Example 36. Let **cons** $d(\mathrm{Y})$, **ask** $\mathrm{Y} > 0$ be a rule guard G. Then

$$ep(G, b(\mathrm{Y}, \mathrm{Z})^2) = d(\mathrm{Y}) \wedge \mathrm{Y} > 0 \wedge b(\mathrm{Y}, \mathrm{Z})^2$$

that is, there must be an atom unifying with the guard and two occurrences of $b(\mathrm{Y}, \mathrm{Z})$ for the corresponding unifying instance of Y. A further example is in Table 7, and refers to rule R_2 of Table 2.

Example 37. Consider rule R of Example 17:

$$ep(R, d(z)) = d(x) \wedge x > 0 \wedge (x = z + 1 \vee d(z)^2 \vee (d(z) \wedge x \neq z))$$

The enabling precondition requires the rule guard to be satisfied: $d(x) \wedge x > 0$. Moreover, the tuple space has to contain

(1) an atom $d(z + 1)$, with $z + 1 > 0$: $d(x) \wedge x > 0 \wedge x = z + 1$; or
(2) two atoms $d(z)$: $d(x) \wedge x > 0 \wedge d(z)^2$; or
(3) one atom $d(z)$, with $z \neq x$: $d(x) \wedge x > 0 \wedge d(z) \wedge x \neq z$.

If none of these condition is satisfied, it is not possible to end up in a state containing $d(z)$.

Remark 38. If we compare the example above with Table 6, condition 2 is the same in the two cases, while conditions 1 and 3 are weaker here:

(1) in case of angelic choice, the presence of an atom $d(z + 1)$ with $z + 1 > 0$ is a sufficient condition, while if the choice is demonic all the atoms with predicate d and argument > 0 are required to equal $d(z + 1)$.
(3) the presence of $d(z)+d(x)$, with $x > 0$ is not a sufficient condition if the choice is demonic: the demon would select $d(z)$. He cannot if $z \leqslant 0$, as required by condition 3 of Table 6.

4.4.2. Properties

The following statements provide some properties of the enabling precondition. All the proofs are in Appendix A.

Proposition 39. *Let p and q be state formulae and R a rule, then*

$$ep(R, p \vee q) \equiv ep(R, p) \vee ep(R, q)$$

Corollary 40. *Let p and q be state formulae and R a rule, then*

$$\neg ep(R, p) \wedge ep(R, q) \rightarrow ep(R, q \wedge \neg p)$$

Corollary 41. *Let p and q be state formulae and R a rule, then*

$$\frac{p \rightarrow q}{ep(R, p) \rightarrow ep(R, q)}$$

4.5. Operational semantics: a transition system

We now define the operational semantics of our language, based on enabling preconditions.

Our definition is not standard: the normal approach would have been to define a transition system in the classical way, i.e., by showing the result of the application of a program statement to a computation state, and then turn the next definition into an equivalence result. Though this can be done, it is not central to the purposes of the paper. Our goal is only to describe the allowed computations of a system: so, we do not need to describe program statements as state transformers.

Definition 42 (*Operational semantics*).

$$T(R)(s) = \{s' \mid \forall q\ \forall \bar{x} \text{ free in } q.\ s' \models q(\bar{x}) \rightarrow s \models ep(R, q(\bar{x}))\}$$

$T(R)$ takes a state s and returns the set of states s' such that rule R, when applied in state s, can lead to state s'.

The next lemma ensures that this is a good definition: the relation $s \models p$ is only defined if all the variables of p are bound.

Lemma 43. *Let R be a rule. A variable* x *is free in* $ep(R, q)$ *(*$wp(R, q)$*) if and only if it is free in q.*

Proof. Variables in $ep(R, q)$ ($wp(R, q)$) are either variables of q or variables of R. The variables of R are universally or existentially quantified in $ep(R, q)$ ($wp(R, q)$), since they all occur in the guard or in the body (see remarks 26 and 35).

4.6. Axiomatic versus operational semantics

The following results relate *ep*, *wp* and operational semantics, and close the gap left open in Section 4.1. In the next section, we will exploit these results to prove the correctness of our refinement calculus. All the proofs are in Appendix A.

Proposition 44. *Let q be a state formula and R a rule, then*

$$wp(R, q) \rightarrow ep(R, q).$$

Lemma 45. *Let R be a rule, then*

$$wp(R, true) \equiv ep(R, true).$$

Proposition 46. *Let R be a rule and q a state formula, then*

$$wp(R, q) \equiv \neg\, ep(R, \neg\, q) \wedge ep(R, true).$$

Proposition 47. *Let s be a state and R a rule, then*

$$s \models wp(R, true) \quad \Leftrightarrow \quad T(R)(s) \neq \emptyset.$$

Corollary 48. *Let s be a state and R a rule, then*

$$s \models ep(R, true) \quad \Leftrightarrow \quad T(R)(s) \neq \emptyset.$$

Theorem 49. *Let R be a rule, s a state, and q a state formula, then* $\forall \bar{x}$ *free in q;*

$$s \models ep(R, q(\bar{x})) \quad \Leftrightarrow \quad \exists s'.\, [s' \in T(R)(s) \ \wedge\ s' \models q(\bar{x})].$$

Theorem 50. *Let R be a rule, s a state, and q a state formula, then $\forall \bar{x}$ free in q:*

$$s \models wp(R, q(\bar{x})) \Leftrightarrow T(R)(s) \neq \emptyset \wedge \forall s'.[s' \in T(R)(s) \rightarrow s' \models q(\bar{x})].$$

5. Refinement

Our refinement calculus is based on the following facts: our specifications are interpreted on computations (Definition 13); we can use the operational semantics to describe the allowed computations of a system; we can exploit the axiomatic semantics to define program properties; the relation between axiomatic and operational semantics stated in the previous section guarantees that using the former we can derive properties, which are correct with respect to the operational behavior.

Definition 51 (*Systems*). We denote by $(\mathbb{R}_!^m, P, Is)$ the system consisting of rules $R_1, \ldots, R_m$, partial order P (defining the rule priorities), and initial state Is.

We formally define computations. For the fairness assumptions we can rely on the results of [32, 53], which define a scheduler ensuring weak fairness in rule selection, and a program transformation technique to achieve strong fairness, are defined. Definition 56 introduces the notion of refinement.

Definition 52 (*Computations*). A computation of a system $(\mathbb{R}_!^m, P, Is)$ is a sequence of states: $s_0, s_1, \ldots, s_i, \ldots$ such that $Is = s_0$ and for all i there exists a rule R_j with

$$s_{i+1} \in T(R_j)(s_i)$$
$$\forall R_k > R_j.\ T(R_k)(s_i) = \emptyset$$

and transition relation $T(R)$ as in Definition 42.

Definition 53 (*Fair computations, with respect to rule selection*). We require that rule selection is fair, i.e. that there does not exists a rule R_j *enabled* in infinite states of c and never *chosen*, where we define a rule R_j to be *enabled* in a state s_i if

$$\exists s' \in T(R_j)(s_i)$$
$$\forall R_k > R_j.T(R_k)(s_i) = \emptyset$$

and to be chosen in s_i if

$$s_{i+1} \in T(R_j)(s_i).$$

Definition 54 (*Fair computations, with respect to tuple selection*). We require tuple selection to be fair as well: no tuple has to be *selectable* infinitely often without ever being *selected*.

We define a tuple $t(\bar{x})$ to be *selectable* in a state s_i if there exist a rule R_j enabled in s_i, a state $s' \in T(R)(s_i)$, and a formula q with $s' \models q$ and $s \not\models q$, such that $ep(R_j, q) \rightarrow t(\bar{x})$.

Given a computation $s_0, s_1, \ldots, s_i, s_{i+1}, \ldots$, we say that $t(\bar{x})$ is *selected* in s_i if $s_{i+1} \in T(R)(s_i)$ for a given R, and there exists a formula q with $s_{i+1} \models q$, $s_i \not\models q$, and $ep(R, q) \rightarrow t(\bar{x})$

Definition 55 (*Fair computations*). A computation is fair when it is fair with respect to rule and tuple selection.

Definition 56 ($(\mathbb{R}_1^m, P, Is) \rhd S$). System $(\mathbb{R}_1^m, P, Is)$ refines a set of properties S, when all the fair computations of $(\mathbb{R}_1^m, P, Is)$ are models of the formulae in S.

To define the calculus we first address the refinement of a specification into a program, and then extend this result to the verification of refinement steps between heterogeneous systems. Besides, we take a compositional approach. We first define what it means for a rule to refine a property, and for an initial state to refine a property, then we show that we can use the axiomatic semantics to derive these relations. Finally, we show how to compose the derived relations to prove that a program refines a set of properties, i.e., a specification.

In the next section we deal with safety properties, and in Section 5.2 with liveness. Finally, in Section 5.3 we introduce heterogeneous systems and propose a methodology of refinement.

5.1. Refinement of safety properties

The common, informal definition for safety is the following: a safety property guarantees that *nothing bad* happens. A formalization of this intuition, based on closed sets, is due to Alpern and Shneider [2]: let a computation c be any infinite sequence of states, and let C be a set of computations, then a property P is a safety property whenever

$$\forall c \in C.[P(c) \leftarrow (\forall c' \text{ finite prefix of } c.P(c'))]$$

Very complex safety properties can be expressed, by means of the σ-*wff* specified in Definition 7. The following is not the worst we can write:

$$\text{INIT } p(\bar{x}) \rightarrow [q(\bar{y}, \bar{z}) \text{ UNLESS } (r(\bar{x}, \bar{y}) \wedge \neg \bigcirc b(\bar{z}))]$$

However, in realistic cases, safety formulae are of the kind INIT p, p UNLESS q, INV p (INIT $p \wedge$ STABLE p), p NEEDS q, with p and q state formulae.

We define $Is \rhd$ INIT p (initial state Is refines INIT p). Then, we define $R \rhd p$ UNLESS q (rule R refines p UNLESS q), $R \rhd p$ NEEDS q, and $R \rhd p$ AFTER q. Proposition 60 relates the weakest precondition calculus to the verification of safety properties, thus proving that refinement relations can be derived exploiting the axiomatic semantics.

We need to introduce the notion of *sticky* variables. Informally, x is sticky in $p(x, y) \rightarrow q(x, z)$, while y and z are not.

Definition 57. Variables $\bar{x}$ are sticky in formula F if $t(F) = \forall \bar{x}.(F' \rightarrow F'')$, with function t as in Definition 10.

Definition 58. $Is \rhd \text{INIT } p$ iff $Is \models p$.

Definition 59. Let p and q be state formulae, then

$R \rhd p \text{ UNLESS } q$ iff $\forall s$, $\forall \bar{x}$ sticky in $p \text{ UNLESS } q$:

$$s \models (p \wedge \neg q) \Rightarrow [T(R)(s) = \emptyset \vee (\forall s'.s' \in T(R)(s) \rightarrow s' \models (p \vee q))]$$

$R \rhd p \text{ NEEDS } q$ iff $\forall s$, $\forall \bar{x}$ sticky in $p \text{ NEEDS } q$:

$$s \models \neg p \Rightarrow (\forall s'.s' \in T(R)(s) \rightarrow s' \models (\neg p \vee q))$$

$R \rhd p \text{ AFTER } q$ iff $\forall s$, $\forall \bar{x}$ sticky in $p \text{ AFTER } q$:

$$s \models (\neg p \wedge \neg q) \Rightarrow \forall s'.s' \in T(R)(s) \rightarrow s' \models \neg p$$

Proposition 60. *Let p and q be state formulae, then*

$$R \rhd p \text{ UNLESS } q \quad \textit{iff} \quad (p \wedge \neg q) \rightarrow (wp(R, p \vee q) \vee \neg wp(R, true))$$
$$R \rhd p \text{ NEEDS } q \quad \textit{iff} \quad ep(R, p \wedge \neg q) \rightarrow p$$
$$R \rhd p \text{ AFTER } q \quad \textit{iff} \quad ep(R, p) \rightarrow (p \vee q)$$

Proof.

- UNLESS: for all states s, $\forall \bar{x}$ *sticky in* $p \text{ UNLESS } q$:

$$s \models (wp(R, p \vee q) \vee \neg wp(R, true))$$
$$\Leftrightarrow$$
$$s \models wp(R, p \vee q) \vee s \models \neg wp(R, true))$$
$$\Leftrightarrow \quad \text{(Theorem 50 and Proposition 47)}$$
$$(\forall s'.s' \in T(R)(s) \Rightarrow s' \models (p \vee q)) \vee (T(R)(s) = \emptyset)$$

- NEEDS: for all states s, $\forall \bar{x}$ *sticky in* $p \text{ NEEDS } q$:

$$s \models \neg p \Rightarrow (\forall s'.s' \in T(R)(s) \Rightarrow s' \models (\neg p \vee q))$$
$$\Leftrightarrow$$
$$(\exists s' \in T(R)(s) \wedge s' \models (p \wedge \neg q)) \Rightarrow s \models p$$
$$\Leftrightarrow \quad \text{(Theorem 49)}$$
$$s \models ep(R, p \wedge \neg q) \Rightarrow s \models p$$

- AFTER: for all states s, $\forall \bar{x}$ *sticky in* p AFTER q:

$$s \models (\neg p \wedge \neg q) \Rightarrow \forall s'.s' \in T(R)(s) \rightarrow s' \models \neg p$$
$$\Leftrightarrow$$
$$(\exists s' \in T(R)(s) \wedge s' \models p) \Rightarrow s \models (p \vee q)$$
$$\Leftrightarrow \quad \text{(Theorem 49)}$$
$$s \models ep(R, p) \Rightarrow s \models (p \vee q) \qquad \square$$

As an example, consider invariants: a property p is invariant (INV) for a system if p holds in the initial state and p is kept invariant during system execution, i.e. if INIT $p \wedge$ STABLE p holds, or, equivalently, INIT $p \wedge p$ UNLESS *false* holds. Then, according to proposition 60, $R \rhd$ INV p if $p \rightarrow (wp(R, p) \vee \neg wp(R, true))$, which precisely states that R will not fire, unless it leaves p untouched.

Example 61. Property $p = d(z) \rightarrow z \geqslant 0$ is kept invariant by

$$R: \textbf{cons } d(x) \textbf{ ask } x > 0 \mid \textbf{ body } y \textit{ is } x - 1. \textbf{ tell } d(y).$$

Indeed, $wp(R, p) \vee \neg wp(R, true)$ equals (Definition 25)

$$(d(x) \wedge x > 0) \rightarrow [(z \neq x-1 \wedge z \neq x \wedge d(z)) \rightarrow z \geqslant 0]$$
$$\vee$$
$$\neg(d(x) \wedge x > 0)$$

which is true whenever $d(z) \rightarrow z \geqslant 0$.

The notion of *refinement under given hypothesis* will be needed in Proposition 64.

Definition 62. Let p, q, and r be state formulae, then

$R \rhd_r$ p UNLESS q iff $\forall s$, $\forall \bar{x}$ sticky in p UNLESS q:

$$s \models (p \wedge r \wedge \neg q) \Rightarrow$$
$$[T(R)(s) = \emptyset \vee (\forall s'.s' \in T(R)(s) \rightarrow s' \models (p \vee q))]$$

$R \rhd_r$ p NEEDS q iff $\forall s$, $\forall \bar{x}$ sticky in p NEEDS q:

$$s \models (r \wedge \neg p) \Rightarrow (\forall s'.s' \in T(R)(s) \rightarrow s' \models (\neg p \vee q))$$

$R \rhd_r$ p AFTER q iff $\forall s$, $\forall \bar{x}$ sticky in p AFTER q:

$$s \models (r \wedge \neg p \wedge \neg q) \Rightarrow (\forall s'.s' \in T(R)(s) \rightarrow s' \models \neg p)$$

Proposition 63. *Let p, q, r be state formulae, then*

$$R \rhd_r p \text{ UNLESS } q \quad \textit{iff} \quad (p \wedge r \wedge \neg q) \rightarrow (wp(R, p \vee q) \vee \neg wp(R, true))$$
$$R \rhd_r p \text{ NEEDS } q \quad \textit{iff} \quad ep(R, p \wedge \neg q) \rightarrow (p \vee \neg r)$$
$$R \rhd_r p \text{ AFTER } q \quad \textit{iff} \quad ep(R, p) \rightarrow (p \vee q \vee \neg r)$$

Proof.

- UNLESS: same proof as for Proposition 60; the same equivalenceis have to be shown, independent of r.
- NEEDS: for all states s, $\forall \bar{x}$ sticky in p UNLESS q:

$$s \models (r \wedge \neg p) \Rightarrow (\forall s'.s' \in T(R)(s) \Rightarrow s' \models (\neg p \vee q))$$
$$\Leftrightarrow$$
$$(\exists s' \in T(R)(s) \wedge s' \models (p \wedge \neg q)) \Rightarrow s \models (p \vee \neg r)$$
$$\Leftrightarrow \qquad \text{(Theorem 49)}$$
$$s \models ep(R, p \wedge \neg q) \Rightarrow s \models (p \vee \neg r)$$

- AFTER: for all states s, $\forall \bar{x}$ *sticky* in p AFTER q:

$$s \models (r \wedge \neg p \wedge \neg q) \Rightarrow \forall s'.s' \in T(R)(s) \rightarrow s' \models \neg p$$
$$\Leftrightarrow$$
$$(\exists s' \in T(R)(s) \wedge s' \models p) \Rightarrow s \models (p \vee q \vee \neg r)$$
$$\Leftrightarrow \qquad \text{(Theorem 49)}$$
$$s \models ep(R, p) \Rightarrow s \models (p \vee q \vee \neg r) \qquad \square$$

To conclude the case of safety properties refinement, we show how to derive the general property that a system refines a set of safety properties.

Proposition 64. *Let $(\mathbb{R}_1^m, P, Is)$ be a system, and S a set of safety properties. We have $(\mathbb{R}_1^m, P, Is) \rhd S$ when, $\forall f \in S$, and $\forall j \in 1 \ldots m$:*

$$Is \rhd f$$
$$R_j \rhd_{H_j} f \quad \textit{with} \quad H_j = \{\neg wp(R_k, true) \mid R_k > R_j \in P\}$$

Proof. According to Definition 56, we have to show that all the fair computations of the system are models for all the formulae in S. We prove the stronger result that all (even the unfair) computations are models for S.

Let $c = Is, s_1, \ldots, s_i, \ldots$ be a computation and f a safety formula in S. We consider the cases $f =$ INIT p and $f = p$ UNLESS q. Proofs for NEEDS and AFTER are similar.

- *Case* $f =$ INIT p.

$$c \textit{ satisfies } f$$
$$\Leftrightarrow \qquad \text{(Remark 9, Definition 14)}$$

$$
\begin{array}{ll}
& Is|\models p \\
\Leftrightarrow & \qquad \text{(Definition 58)} \\
& Is \rhd \text{INIT } p
\end{array}
$$

- *Case* $f = p$ UNLESS q. By induction on the length of the computation.
  - Base. Let $c_0 = Is, Is \ldots$, we need to show that (Definitions 8)

    $$Is|\models p \wedge \neg q \Rightarrow Is|\models p \vee q$$

    and this is always true.
  - Inductive step. Let $c_i = Is, s_1, \ldots, s_i, s_i, s_i, \ldots$ satisfy f, we show that $c_{i+1} = Is, s_1, \ldots, s_i, s_{i+1}, s_{i+1}, s_{i+1}, \ldots$ satisfy f as well. We need to show that

    $$s_i|\models p \wedge \neg q \Rightarrow s_{i+1}|\models p \vee q$$

    Let $s_{i+1} \in T(R_k)(s_i)$ for some k. By Definition 52, Proposition 47, and Corollary 48, we have

    $$
    \begin{array}{ll}
    & s_i|\models ep(R_k, true) \\
    & s_i|\not\models ep(R_l, true) \quad \forall R_l.\, R_l > R_k \text{ in } P \qquad \text{that is: } s_i|\models H_k
    \end{array} \tag{8}
    $$

  Now

  $$
  \begin{array}{ll}
  & R_k \rhd_{H_k} \; p \text{ UNLESS } q \\
  \Leftrightarrow & \qquad \text{(Definition 62)} \\
  & \forall s.\, s \models (p \wedge H_k \wedge \neg q) \Rightarrow \\
  & \quad [T(R_k)(s) = \emptyset \vee (\forall s'.\, s' \in T(R_k)(s) \rightarrow s' \models (p \vee q))]
  \end{array}
  $$

We take $s = s_i$ and $s' = s_{i+1}$, we have $T(R_k)(s_i) \neq \emptyset$ and $s_{i+1} \in T(R_k)(s_i)$ by construction, hence

$$s_i \models (p \wedge H_k \wedge \neg q) \Rightarrow s_{i+1} \models (p \vee q))$$

Since (see 8) $s_i|\models H_k$, we conclude

$$s_i \models (p \wedge \neg q) \Rightarrow s_{i+1} \models (p \vee q)) \qquad \square$$

Let us use again the Unity operator INV, where INV p denotes the conjunction of INIT p and STABLE p. We show, for instance, how to derive that a property p is invariant for system $(\mathbb{R}_1^m, \emptyset, Is)$, by proving that the initial state refines p and that $R_i \rhd_{H_i}$ STABLE p for every rule R_i.

Example 65. Let p be a state formula, and let $(\mathbb{R}_1^m, \emptyset, Is)$ be a system

$$
\begin{array}{lll}
R_1 \rhd \text{STABLE } p \ \ldots \ R_m \rhd \text{STABLE } p & \Rightarrow & \mathbb{R}_1^m \rhd \text{STABLE } p \\
\mathbb{R}_1^m \rhd \text{STABLE } p \text{ and } Is \rhd \text{INIT } p & \Rightarrow & (\mathbb{R}_1^m, \emptyset, Is) \rhd \text{INV } p
\end{array}
$$

5.2. Refinement of liveness properties

The class of liveness properties has a characterization as well. Informally, liveness guarantees that *something good eventually* happens. Formally, let C be any set of computations (generic infinite sequences of states), then a property P is a liveness property whenever [2]:

$$\forall\, c'.\text{finite prefix of a } c \in C\, \exists c'' \text{ infinite extension of } c'.P(c'')$$

We show here how to prove that a system refines a liveness property. Let us first observe that the distinction between the three progress operators UNTIL, $\mapsto$, and $\hookrightarrow$, is useful in the first steps of a refinement, when we refine specifications (for instance $\mapsto$ is transitive while UNTIL is not), but it is meaningless when we introduce rules: in any case we have to show that an UNTIL property is refined. Indeed, in our setting, a rule only relates pairs of successive states: if a rule does *something good*, then it does it in a computation step. Then, if a step leads from a state satisfying p to a state satisfying q, it means that the strongest liveness property, p UNTIL q, is satisfied.

We hence show how to prove that a system refines an UNTIL property, and apply the same results when refining the weaker $\mapsto$ or $\hookrightarrow$.

We first define $R \rhd p \text{ UNTIL } q$ (rule R refines p UNTIL q), and then show how to derive this relation exploiting the axiomatic semantics, with a result relating our enabling precondition calculus with program verification. The following definition only captures the *progress* part of an UNTIL property. Proposition 68 adds the needed safety conditions.

Definition 66. Let p, q be state formulae, $\bar{x}$ sticky in p UNTIL q, then

$$R \rhd p \text{ UNTIL } q \quad \text{iff} \quad \forall s\, \forall \bar{x} :\ s \models (p \wedge \neg q) \Rightarrow \exists s'.[s' \in T(R)(s) \wedge s' \models q]$$

Proposition 67. *Let p, q be state formulae, $\bar{x}$ sticky in p UNTIL q, then*

$$R \rhd p \text{ UNTIL } q \quad \textit{iff} \quad \forall \bar{x}.\, [(p \wedge \neg q) \rightarrow ep(R, q)]$$

Proof. For all states s,

$$\begin{aligned} & s \models ep(R, q) \\ & \quad \Leftrightarrow \qquad \text{(Theorem 49)} \\ & \exists s'.[s' \in T(R)(s) \wedge s' \models q] \end{aligned}$$

□

We recall that a computation $c = s_0, \ldots, s_i, \ldots$ satisfies p UNTIL q when, for all s_i, $s_i \models p$ implies $s_{i+1} \models p \vee q$ and there exists j such that $s_{i+j} \models q$. The following result shows how to prove that a system refines an UNTIL property.

Proposition 68. *Let p and q be state formulae, $(\mathbb{R}_1^m, P, Is)$ be a system, and for some i :*

(1) $R_i \rhd p \text{ UNTIL } q$

(2) $(\mathbb{R}_1^m \setminus \{R_j < R_i\}, P, Is) \rhd p$ UNLESS q
(3) $(\{R_k > R_i\}, P, Is) \rhd wp(R_j, true)$ UNTIL $\neg wp(R_j, true)$, $\forall R_j > R_i$.
Then $(\mathbb{R}_1^m, P, Is) \rhd p$ UNTIL q.

Proof. In the general case, in order for a system to refine an UNTIL property, we have to guarantee, see Definition 56, that at least a rule R refines the *progress* (1), and that R and all the other rules satisfy the corresponding UNLESS property (2). If there is a priority order between rules, on one side we only have to require the system containing rules having equal or greater priority than R to satisfy such a safety constraint, on the other side we have to require that R will fire sometimes, i.e. that a state is reached in which $wp(R_j, true) = false$, being R_j the rules having greater priority than R (3). □

In the special case of empty priority set the proposition above reduces to:

Corollary 69. *Let p, q be state formulae, a system $(\mathbb{R}_1^m, \emptyset, Is) \rhd p$ UNTIL q if for some i :*
(1) $R_i \rhd p$ UNTIL q,
(2) $R_j \rhd p$ UNLESS q *for all* R_j, $j = 1, \ldots, m$.

Example 70. Let s be the state $\{n(1), n(2), n(3)\}$, and L the liveness property

$$L: \; n(X) + n(Y) \wedge X = (Y + 1) \quad \mapsto \quad \neg n(X)$$

any computation of a system refining L must lead to state $\{n(1)\}$. Let us now consider the rule

$$R : \textbf{cons } n(X) \textbf{ ask } n(Y) \textbf{ ask } X = Y + 1 \mid$$

R satisfies the hypothesis of Proposition 67, but it does not refine

$$S: \; n(X) + n(Y) \wedge X = (Y + 1) \quad \text{UNLESS} \quad \neg n(X)$$

Consequently, system $\Sigma = (\{R\}, \emptyset, s)$ does not refine L (condition 2 of Corollary 69 is not satisfied). Indeed, if $n(2)$ is consumed first, then Σ stops in state $\{n(1), n(3)\}$. An alternative rule, refining both L and S is the following:

$$R' : \textbf{cons } n(X) \textbf{ ask } n(Y) \textbf{ ask } X = Y + 1 \textbf{ ask } Z > X \textbf{ ask not } n(Z) \mid$$

5.3. Heterogeneity

Heterogeneity deals with the intermediate steps of a step-wise derivation, the *unrefined programs*, which can be written in the same language of the final program, as in [52], in a specification language, as in [24, 25, 35, 48], or in a mix of the two, as advocated in [1]. In the first case we have the advantage that we can execute and test the unrefined programs, in the second case we can refine the system abstracting from

the low-level details of a programming language. Finally, in the third, heterogeneous case, we can concentrate on the refinement of a part of the program assuming that the others (usually referred to as the *environment*) behave correctly.

We permit to exploit logic formulae to specify the behavior of unrefined programs. Each intermediate step is partially written in a tuple space language, and some unrefined features are described by logical formulae. In the heterogeneous view, a system consists of safety properties, liveness properties, rules, a priority order, and an initial state: $(S, L, \mathbb{R}_1^m, P, Is)$. Each of these components can be empty. In the first refinement step $m = 0$, $P = \emptyset$, and $Is = \emptyset$, i.e., we start from a specification, in the last one $L = \emptyset$, i.e., we end up into an executable system. A system $(S, L, \mathbb{R}_1^m, P, Is)$ has to satisfy some constraints, in particular $\mathbb{R}_1^m$ have to preserve the safety constraints in S. Such a condition gives us the ability to compose these systems.

Definition 71 (*$s(L)$, safe-liveness properties*). Let L be a set of liveness properties, we call $s(L)$ the set

$$\{p \text{ UNLESS } q \mid p \text{ UNTIL } q,\ p \mapsto q \text{ or } p \hookrightarrow q \in L\}$$

Definition 72 (*Heterogeneous system*). A system $(S, L, \mathbb{R}_1^m, P, Is)$ satisfies
(1) $Is \rhd S$
(2) $\mathbb{R}_1^m \rhd S \cup s(L)$

In other words, if at a given refinement step, our system is a heterogeneous one, say: $(S, L, \mathbb{R}_1^m, P, Is)$, we require not only the *environment* to satisfy properties in S and L, but we also impose some constraints to the rules we already defined. In particular, in order for $(S, L, \mathbb{R}_1^m, P, Is)$ to be an heterogeneous system, we require rule $\mathbb{R}_1^m$ to refine both the safety properties in S and the safe-liveness properties in $s(L)$.

Definition 73 (*Computations of heterogeneous systems*). A computation $c = s_0, s_1, \ldots,$ of a heterogeneous system $\Sigma = (S, L, \mathbb{R}_1^m, P, Is)$ is a model for the formulae in $S \cup L$, satisfies fairness with respect to the transition system defined by rules $\mathbb{R}_1^m$, and satisfies $Is \subseteq s_0$, (with $\subseteq$ multiset inclusion) and $s_0 \rhd S$. We call $C(\Sigma)$ the set of computations of Σ.

Definition 74 (*Systems refinement*). Let Σ and Σ' be systems (either heterogeneous or executable), $\Sigma' \rhd \Sigma$ iff $C(\Sigma') \subseteq C(\Sigma)$.

The next statement introduces a refinement pattern (refinement by *fulfillment*) for heterogeneous systems, permitting to cancel liveness properties that are fulfilled by the introduced rules, and to substitute them with suitable safety properties. We can read it in the following perspective. A heterogeneous system refines another one if new rules are introduced to "give a help", i.e. to satisfy liveness. In some sense we can look at these new rules as to the *environment* of the rules of the unrefined system.

Proposition 75 (Refinement by fulfillment). *Let* $(S, L, \mathbb{R}_1^m, P, Is)$ *be a heterogeneous system,* L', L'' *liveness sets, such that* $L = L' \cup L''$, R *be a rule, and let the partial order* P' *extend* P *to accommodate* R. *Then*

$$(S \cup s(L'), L'', \mathbb{R}_1^m \| R, P', Is) \rhd (S, L, \mathbb{R}_1^m, P, Is)$$

provided that
(1) $R \rhd L'$
(2) $R \rhd_H S \cup s(L') \cup s(L'')$
(3) $(\{R_k \mid R_k \geqslant R\}, P', Is) \rhd wp(R_j, true)$ UNTIL $\neg wp(R_j, true) \quad \forall R_j \geqslant R$
with $H = \{\neg wp(R_k, true) \mid R_k > R \in P'\}$.

Proof. We call $\Sigma' = (S \cup s(L'), L'', \mathbb{R}_1^m \| R, P', Is)$ and $\Sigma = (S, L, \mathbb{R}_1^m, P, Is)$. We need to prove two points:
(1) Σ' is a heterogeneous system (Definition 72)

Σ is a heterogeneous system
$\Rightarrow$
$\mathbb{R}_1^m \rhd S \cup s(L)$
$\Leftrightarrow \quad (L = L' \cup L'')$
$\mathbb{R}_1^m \rhd S \cup s(L') \cup s(L'')$
$\Rightarrow$ (By hypothesis 2: $R \rhd S \cup s(L') \cup s(L'')$, Proposition 64)
$\mathbb{R}_1^m \| R \rhd S \cup s(L') \cup s(L'')$

Moreover, $Is \rhd S$ since Σ is an heterogeneous system, and the UNLESS properties, like those in $s(L')$, are satisfied by an initial state. Consequently, $Is \rhd S \cup s(L')$

(2) $C(\Sigma') \subseteq C(\Sigma)$. By contradiction, let $c = s_0, s_1, \ldots, s_i, \ldots$ be a computation, with $c \in C(\Sigma')$ and $c \notin C(\Sigma)$. Then (Definition 73) one of the following conditions holds:
(a) c is not a model for S;
(b) c is not a model for L;
(c) c does not satisfy fairness with respect to $\mathbb{R}_1^m$
(d) $Is \not\subseteq s_0$
(e) $s_0 \not\rhd S$

Conditions (a), (c)–(e) are immediately falsified since c is a computation of the heterogeneous system Σ'. Condition (b) implies that there is (at least) a liveness property F in L which is not satisfied by c: since $L = L' \cup L''$, either $F \in L''$ or $F \in L'$. According to Definition 73, it is not possible that $F \in L''$. We are left with $F \in L'$. This is also impossible, since by hypothesis (1) $R \rhd L'$, and R is guaranteed to fire if its guard becomes true by hypothesis (3) and the fairness assumption. Finally, $\mathbb{R}_1^m \rhd s(L')$ guarantees that the other rules do not interfere with the task of R in refining formulae in L'. □

In the special case of empty priority set the proposition above reduces to:

Corollary 76.
$(S \cup s(L'), L'', \mathbb{R}_1^m \| R, \emptyset, Is) \rhd (S, L, \mathbb{R}_1^m, \emptyset, Is)$, *provided that*
(1) $R \rhd L'$
(2) $R \rhd S \cup s(L') \cup s(L'')$

Working with heterogeneous systems encourages a methodology of refinement: starting from a specification containing *liveness* and *safety* properties, a refinement step is guided by liveness (we introduce rules "doing something good"), and must satisfy safety (the introduced rules "don't have to do anything bad"). When all the liveness properties are guaranteed, we are left with system $(S, \emptyset, \mathbb{R}_1^m, P, Is)$ where all the safety properties in S are satisfied by $\mathbb{R}_1^m$. Proposition 79 shows that the executable system $(\mathbb{R}_1^m, P, Is)$ refines it.

The constraints we impose on heterogeneous systems make our approach different from the approaches to *modular refinement* based on a stronger distinction between local system and environment. For instance, in the *rely–guarantee* approach, Jones makes a clear distinction between properties that the local state has to *guarantee* and properties on which we can *rely* [38]. This generates some problems when trying to compose different systems. The topic of composition in this framework is well described by Abadi and Lamport [1].

The following two propositions introduce other refinement patterns: *specification* and *initial state* refinement, respectively.

Proposition 77 (Specification refinement). *Let* $(S, L, \mathbb{R}_1^m, P, Is)$ *be a heterogeneous system, and let* S', L' *be sets of safety and liveness properties, respectively, such that*
(1) $S' \cup L' \rightarrow S \cup L$,
(2) $\mathbb{R}_1^m \rhd (S' \setminus S) \cup s(L' \setminus L)$,
(3) $Is \rhd S' \setminus S$
Then $(S', L', \mathbb{R}_1^m, P, Is) \rhd (S, L, \mathbb{R}_1^m, P, Is)$.

Proof. We call $\Sigma = (S, L, \mathbb{R}_1^m, P, Is)$ and $\Sigma' = (S', L', \mathbb{R}_1^m, P, Is)$. We need to prove two points:
(1) Σ' is an heterogeneous system (Definition 72)

$$\mathbb{R}_1^m \rhd (S' \setminus S) \cup s(L' \setminus L) \qquad \text{(by hypothesis)}$$
$$\mathbb{R}_1^m \rhd S \cup s(L) \qquad \text{(since } \Sigma \text{ is an heterogeneous system)}$$
$$\Rightarrow$$
$$\mathbb{R}_1^m \rhd S' \cup s(L')$$

$$Is \rhd (S' \setminus S) \qquad \text{(by hypothesis)}$$

$Is \rhd S$ (since Σ is an heterogeneous system)

$\Rightarrow$

$Is \rhd S'$

(2) $C(\Sigma') \subseteq C(\Sigma)$: let $c = s_0, s_1, \ldots, s_i, \ldots \in C(\Sigma')$, we show that $c \in C(\Sigma)$:
   (a) c is a model for $S' \cup L'$, $S' \cup L' \rightarrow S \cup L$, then c is a model for $S \cup L$;
   (b) the fairness condition is guaranteed since the set of rules is unchanged in the two systems;
   (c) $Is \subseteq s_0$ is guaranteed since Is is unchanged in the two systems;
   (d) $s_0 \rhd S'$ and $S' \rightarrow S$ imply $s_0 \rhd S$. □

Proposition 78 (Initial state refinement). *Let* $(S, L, \mathbb{R}_1^m, P, Is)$ *be a heterogeneous system, and let* Is' *be an initial state such that*
(1) $Is' \supseteq Is$
(2) $Is' \rhd S$
Then $(S, L, \mathbb{R}_1^m, P, Is') \rhd (S, L, \mathbb{R}_1^m, P, Is)$.

Proof. We call $\Sigma = (S, L, \mathbb{R}_1^m, P, Is)$ and $\Sigma' = (S, L, \mathbb{R}_1^m, P, Is')$. We need to prove two points:
(1) Σ' is a heterogeneous system: only the initial state has been changed and, by hypothesis, $Is' \rhd S$.
(2) $C(\Sigma') \subseteq C(\Sigma)$: let $c = s_0, s_1, \ldots, s_i, \ldots \in C(\Sigma')$, then (Definition 73) $Is' \subseteq s_0$, and $s_0 \rhd S$. By hypothesis $Is' \supseteq Is$, hence $c \in C(\Sigma)$. □

Proposition 79. *Let* $\Sigma = (S, \emptyset, \mathbb{R}_1^m, P, Is)$ *be a heterogeneous system, and let* $\Sigma' = (\mathbb{R}_1^m, P, Is)$ *be the corresponding executable system. Then* $\Sigma' \rhd \Sigma$

Proof. We have to prove that $C(\Sigma') \subseteq C(\Sigma)$ (Definition 74). Let $c \in C(\Sigma')$, according to Definitions 52 and 55, c satisfies fairness with respect to the transition system defined by rules $\mathbb{R}_1^m$. We know that Σ is an heterogeneous system, hence any computation in the transition system defined by $\mathbb{R}_1^m$ is a model for S, and $Is \rhd S$. Then, by Definition 73, $c \in C(\Sigma)$. □

Corollary 80. *Let* $\Sigma = (\mathbb{R}_1^m, P, Is)$ *be an executable system, obtained by refinement steps (applying Propositions* 75, 77–79*) from* $(S, L, \emptyset, \emptyset, \emptyset)$, *then every computation of* Σ *refines the liveness properties in* L, *and preserves the safety conditions in* S.

Finally, we show how to compose heterogeneous systems.

Proposition 81 (Composing heterogeneous systems). *Let* $(S', L_1' \cup L_2', \emptyset, \emptyset, \emptyset)$ *be a specification,* $\mathbb{R}_1^m$ *and* $\mathbb{R}_{1+m}^n$ *disjoint sets of rules, and let*
(1) $(S, L_1, \mathbb{R}_1^m, P_1, Is_1) \rhd (S' \cup s(L_2'), L_1', \emptyset, \emptyset, \emptyset)$
(2) $(S, L_2, \mathbb{R}_{1+m}^n, P_2, Is_2) \rhd (S' \cup s(L_1'), L_2', \emptyset, \emptyset, \emptyset)$
(3) $Is_1 \cup Is_2 \rhd S'$

Then

$(S, L_1 \cup L_2, \mathbb{R}^n_1, P_1 \cup P_2, Is_1 \cup Is_2) \rhd (S', L'_1 \cup L'_2, \emptyset, \emptyset, \emptyset)$

Proof. The statement directly applies Proposition 75 (in a simplified version, since here the priority sets are disjoint), and Proposition 77. □

6. An example: Eratosthenes' sieve

We start with the problem specification, using the following encoding:
primes(P) : – a request to compute primes up to P has been issued.
prime(X) : – X is prime.
We assume $P > 2$.
System E_0 states that all and only prime numbers will be computed:

E_0 : L : $primes(P) \wedge 2 \leqslant X \leqslant P \wedge \neg(2 \leqslant Y < X \wedge Y|X) \hookrightarrow prime(X)$
S : $prime(X)$ NEEDS $primes(P) \wedge 2 \leqslant X \leqslant P \wedge \neg(2 \leqslant Y < X \wedge Y|X)$

◁ We have to split L, to introduce the generator and the sieve. We also have to split safety accordingly. L_1 says that we need consider all the eligible numbers. L_2 reconstructs L, according to C6 (Table 5) and

$$primes(P) \wedge 2 \leqslant X \leqslant P \quad \hookrightarrow \quad n(X)$$

$$\Rightarrow$$

$$primes(P) \wedge 2 \leqslant X \leqslant P \wedge \neg(2 \leqslant Y < X \wedge Y|X) \hookrightarrow n(X) \wedge \neg(2 \leqslant Y < X \wedge Y|X)$$

With respect to safety, S_1 states that numbers are generated only for the sieve, S_2 that a prime $prime(X)$ is computed only if X is prime and the corresponding number $n(X)$ has already been generated. Finally S_3 says that the request to compute primes has to be maintained. Refinement correctness for safety is ensured by N6 (Table 5) and

$$n(X) \quad \text{NEEDS} \quad primes(P) \wedge 2 \leqslant X \leqslant P$$

$$\Rightarrow$$

$$n(X) \wedge \neg(2 \leqslant Y < X \wedge Y|X) \text{ NEEDS } primes(P) \wedge 2 \leqslant X \leqslant P \wedge \neg(2 \leqslant Y < X \wedge Y|X)$$

E_1 : L_1 : $primes(P) \wedge 2 \leqslant X \leqslant P \quad \hookrightarrow \quad n(X)$
L_2 : $n(X) \wedge \neg(2 \leqslant Y < X \wedge Y|X) \quad \mapsto \quad prime(X)$
S_1 : $n(X)$ NEEDS $primes(P) \wedge 2 \leqslant X \leqslant P$
S_2 : $prime(X)$ NEEDS $n(X) \wedge \neg(2 \leqslant Y < X \wedge Y|X)$
S_3 : STABLE $primes(P)$

◁ We do not want the sieve to explicitly compute primes, but to cancel non-prime numbers: we encode $prime(X)$ as being in the space once "done": $prime(X) \Leftarrow n(X) + done$, where *done* says that only primes are in the state. We encode

with $max(\mathrm{P})$ the condition that all the primes up to P are in the state. At this stage of the refinement, we assume this condition to be angelically written in the state at some point in the computation (L_3).
We have

$$done + primes(\mathrm{P}) + max(\mathrm{P}) + n(\mathrm{X}) \wedge 2 \leqslant \mathrm{X} \leqslant \mathrm{P} \Rightarrow \neg(2 \leqslant \mathrm{Y} < \mathrm{X} \wedge \mathrm{Y} \mid \mathrm{X})$$

$$done + primes(\mathrm{P}) + max(\mathrm{P}) \wedge 2 \leqslant \mathrm{X} \leqslant \mathrm{P} \wedge \neg(2 \leqslant \mathrm{Y} < \mathrm{X} \wedge \mathrm{Y} \mid \mathrm{X}) \Rightarrow n(\mathrm{X})$$

i.e., all and only primes are in the space. Moreover, we have

$$max(\mathrm{P}) + primes(\mathrm{P}) \wedge 2 \leqslant \mathrm{X} \leqslant \mathrm{P} \wedge \neg(2 \leqslant \mathrm{Y} < \mathrm{X} \wedge \mathrm{Y} \mid \mathrm{X}) \Rightarrow n(\mathrm{X})$$

while the converse is not true, i.e. we do not force the presence of all the primes up to (P) to imply $max(\mathrm{P})$. We refine S_2 according to this new encoding. S_4 says that no new number can be generated after $max(\mathrm{P})$, S_5 says that primes cannot be lost, and S_6 that $max(\mathrm{P})$ is kept unless *done*.

E_2 :
L_1 : $primes(\mathrm{P}) \wedge 2 \leqslant \mathrm{X} \leqslant \mathrm{P} \hookrightarrow n(\mathrm{X})$
L_2 : $n(\mathrm{X}) \wedge 2 \leqslant \mathrm{Y} < \mathrm{X} \leqslant \mathrm{P} \wedge \mathrm{Y} \nmid \mathrm{X} \mapsto n(\mathrm{X}) + done$
L_3 : $\Diamond\ max(\mathrm{P})$
S_1, S_3 : =
S_2 : $done$ NEEDS $primes(\mathrm{P}) + max(\mathrm{P}) + \neg(n(\mathrm{X}) \wedge 2 \leqslant \mathrm{Y} < \mathrm{X} \wedge \mathrm{Y} \mid \mathrm{X})$
S_4 : $n(\mathrm{X})$ NEEDS $\neg\, max(\mathrm{P})$
S_5 : $n(\mathrm{X})$ UNLESS $2 \leqslant \mathrm{Y} < \mathrm{X} \wedge \mathrm{Y} \mid \mathrm{X}$
S_6 : $max(\mathrm{P})$ UNLESS $done$

◁ We refine the sieve by explicitly stating that non-prime numbers must be eliminated ($L_{2.1}$), and that *done* must appear once all and only the primes are left ($L_{2.2}$). Refinement correctness is guaranteed by the safety properties and L_3.

E_3 :
L_1, L_3 : =
$L_{2.1}$: $n(\mathrm{X}) \wedge 2 \leqslant \mathrm{Y} < \mathrm{X} \wedge \mathrm{Y} \mid \mathrm{X} \hookrightarrow \neg n(\mathrm{X})$
$L_{2.2}$: $primes(\mathrm{P}) + max(\mathrm{P}) + \neg(n(\mathrm{X}) \wedge 2 \leqslant \mathrm{Y} < \mathrm{X} \wedge \mathrm{Y} \mid \mathrm{X}) \mapsto done$
$S_1, S_2, S_3, S_4, S_5, S_6$: =

◁ To refine $L_{2.1}$ we introduce rule

$$R_2 : \textbf{cons}\ n(\mathrm{X})\ \textbf{ask}\ n(\mathrm{Y})\ \textbf{ask}\ \mathrm{Y}|\mathrm{X}\ \textbf{ask}\ \mathrm{X} > \mathrm{Y}|$$

which better captures the expected behavior of the sieve than the following immediate refinement of $L_{2.1}$:

$$\textbf{cons}\ n(\mathrm{X})\ \textbf{ask}\ 2 \leqslant \mathrm{Y} < \mathrm{X}\ \textbf{ask}\ \mathrm{Y}|\mathrm{X}|$$

A non-prime $n(\mathrm{X})$ is canceled every time a pair $n(\mathrm{X})$, $n(\mathrm{Y})$ is found, with $\mathrm{Y} \mid \mathrm{X}$, and $\mathrm{X} > \mathrm{Y}$. Note that given $n(\mathrm{X})$ and $2 \leqslant \mathrm{Y} < \mathrm{X}$ with, $\mathrm{Y}|\mathrm{X}$, it can happen that $n(\mathrm{Y})$ is canceled before $n(\mathrm{X})$, ending up in a situation similar to that of Example 70. However, in this case we are ensured (S_5) that an $n(\mathrm{Z})$ exists in the space, with Z prime, that can be used to cancel $n(\mathrm{X})$.

We also refine $L_{2.2}$: to avoid the difficulty with the negation, since the premise of $L_{2.2}$ implies the negation of that of $L_{2.1}$, we exploit priorities, introducing

$$R_3 : \textbf{ask } primes(\mathrm{P}) \textbf{ ask } max(\mathrm{P}) \textbf{ ask not } done \mid \textbf{tell } done$$

$$\textbf{and } R_2 > R_3$$

We add **ask not** *done* since one is enough. Refinement proofs are in Section 6.1. According to Proposition 75, we have to add the safe-liveness properties

$$s(L_{2.1}) : \; n(\mathrm{X}) \wedge 2 \leqslant \mathrm{Y} < \mathrm{X} \wedge \mathrm{Y} \mid \mathrm{X} \quad \text{UNLESS} \quad \neg n(\mathrm{X})$$

$$(L_{2.2}) : \; primes(\mathrm{P}) + max(\mathrm{P}) + \neg done + \neg (n(\mathrm{X}) \wedge 2 \leqslant \mathrm{Y} < \mathrm{X} \wedge \mathrm{Y} \mid \mathrm{X}) \quad \text{UNLESS} \quad done$$

but the first one is a tautology, and the second is implied by S_3, S_6, and S_4.

E_4 : $L_1, L_3 : =$
$S_1, S_2, S_3, S_4, S_5, S_6 : =$
$\{R_2, R_3\}, \{R_2 > R_3\}$

◁ We could refine L_1 by introducing rule

$$R : \textbf{ask } primes(\mathrm{P}) \textbf{ ask } \mathrm{X} \leqslant \mathrm{P} \textbf{ ask not } n(\mathrm{X}) \textbf{ ask not } max(\mathrm{P}) \mid \textbf{tell } n(\mathrm{X})$$

but it would be highly inefficient, since non-prime numbers would be continuously restored. Besides, this rule would not "implement" the angelic behavior that we have assumed with respect to max(P). We hence refine E_4 in order to generate all the numbers orderly and once, according to the following specification refinement:

$$L_1 : \; primes(\mathrm{P}) \wedge 2 \leqslant \mathrm{X} \leqslant \mathrm{P} \quad \hookrightarrow \quad n(\mathrm{X})$$

$$\Leftarrow$$

$$L_{1.1} : \; primes(\mathrm{P}) + max(\mathrm{M}) \wedge \mathrm{M} < \mathrm{P} \quad \text{UNTIL} \quad max(\mathrm{M}+1)$$

$$S_7 : \; max(\mathrm{M}) \quad \text{NEEDS} \quad n(\mathrm{M}) \wedge 2 \leqslant \mathrm{M} \leqslant \mathrm{P}$$

$$S_8 : \; \Box \neg (n(\mathrm{X}) + n(\mathrm{X}))$$

$$S_9 : \; \text{INIT} \quad max(2)$$

Moreover, since $\Diamond\ max(\mathrm{P})$ follows from $L_{1.1}$, also L_3 is refined by $L_{1.1}$

E_5 : $L_{1.1} : \; primes(\mathrm{P}) + max(\mathrm{M}) \wedge \mathrm{M} < \mathrm{P} \quad \text{UNTIL} \quad max(\mathrm{M}+1)$
$S_1, S_2, S_3, S_4, S_5, S_6 : =$
$S_7 : \; max(\mathrm{M}) \quad \text{NEEDS} \quad n(\mathrm{M}) \wedge 2 \leqslant \mathrm{M} \leqslant \mathrm{P}$
$S_8 : \; \Box \neg (n(\mathrm{X}) + n(\mathrm{X}))$
$S_9 : \; \text{INIT} \quad max(2)$
$\{R_2, R_3\}, \{R_2 > R_3\}$

$\lhd$ We can now refine $L_{1.1}$ by introducing the last rule (see also Section 6.2)

R_1 : **cons** $max(\mathrm{M})$ **ask** $primes(\mathrm{P})$ **ask** $\mathrm{M} < \mathrm{P}$
|**body** M' *is* $\mathrm{M} + 1$
tell $= max(\mathrm{M}')$ **tell** $n(\mathrm{M}')$

E_6 : $S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9 :=$
$s(L_{1.1})$: $primes(\mathrm{P}) + max(\mathrm{M}) \wedge \mathrm{M} < \mathrm{P}$ UNLESS $max(\mathrm{M}+1)$
$\{R_1, R_2, R_3\}, \{R_2 > R_3\}$

$\lhd$ To fix the initial state, we need an initial state refinement, and apply Proposition 78, with $Is_E = \{max(2), n(2), primes(\mathrm{P})\}$, with $\mathrm{P} \geqslant 2$

E_7 : $S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, s(L_{1.1}) :=$
$\{R_1, R_2, R_3\}, \{R_2 > R_3\}, Is_E$

$\lhd$ We can finally simplify R_3 exploiting priorities:

R_3 : **ask not** $done$ | **tell** $done$

and $R_1 > R_3$

E_7 : $S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, s(L_{1.1}) :=$
$\{R_1, R_2, R_3\}, \{R_1 > R_3, R_2 > R_3\}, \{max(2), n(2)\}$

$\lhd$ We can now end up in the executable system of Table 2, by Proposition 79

Σ : $\{R_1, R_2, R_3\}, \{R_1 > R_3, R_2 > R_3\}, \{max(2), n(2)\}$

6.1. *Proof of the fourth refinement step:* $E_3 \lhd E_4$

To prove that $E_3 \lhd E_4$, we apply twice Proposition 75, introducing first R_2 and then R_3. To prove that the introduction of R_2 to fulfill $L_{2.1}$ is correct, we have to prove that R_2 refines: $L_{2.1}$, $s(L_1)$, the safety properties in E_4, and $wp(R_2, true)$ UNTIL $\neg wp(R_2, true)$.

Similarly, to prove that the introduction of R_3 is correct, we have to prove that R_3 refines $L_{2.2}$ and $wp(R_3, true)$ UNTIL $\neg wp(R_3, true)$, and that it refines $s(L_1)$ and the safety properties in E_4 under the hypothesis $H_3 = \{\neg wp(R_2, true)\}$ (Proposition 64).

We show a couple of them.

$R_2 \rhd L_{2.1}$ We apply Proposition 67:

In Table 7 we exploit S_8 and prove that

$$(n(\mathrm{X}) + n(\mathrm{Y}) \wedge \mathrm{Y} \,|\, \mathrm{X} \wedge \mathrm{Y} < \mathrm{X}) \rightarrow ep(R_2, \neg n(\mathrm{X}))$$

we now thus left with

$$n(\mathrm{X}) \wedge 2 \leqslant \mathrm{Y} < \mathrm{X} \wedge \mathrm{Y} \,|\, \mathrm{X} \quad \mapsto \quad n(\mathrm{X}) + n(\mathrm{Z}) \wedge \mathrm{Z} \,|\, \mathrm{X} \wedge \mathrm{Z} < \mathrm{X}$$

which exploits S_5 (primes are left in the space) and conclude applying C6 of Table 5.

$R_3 \rhd L_{2.2}$ We apply Proposition 67:

We have to show that

$$primes(\mathrm{P}) + max(\mathrm{P}) + \neg(n(\mathrm{X}) + n(\mathrm{Y}) \ \wedge \ \mathrm{Y} \mid \mathrm{X}) + \neg done \ \rightarrow \ ep(R_3, done)$$

This is true, since calculation shows that $ep(R_3, done)$ coincides with the premise.

6.2. *Proof of the sixth refinement step:* $E_5 \lhd E_6$

To prove $E_5 \lhd E_6$, we apply again Proposition 75. We have to prove that R_1 refines: $L_{1.1}$, the safety properties in E_6, and $wp(R_1, true)$ UNTIL $\neg wp(R_1, true)$; also, we have to prove that R_2 and R_3 refine $s(L_{1.1})$. We show some of them.

$R_1 \rhd L_{1.1}$ We apply Proposition 67:

We have to show that

$$primes(\mathrm{P}) + max(\mathrm{M}) \wedge \mathrm{M} < \mathrm{P} \ \rightarrow \ ep(R_1, max(\mathrm{M}+1))$$

which is true, since

$$ep(R_1, max(\mathrm{M}+1)) = primes(\mathrm{P}) + max(\mathrm{M}) \wedge \mathrm{M} < \mathrm{P}$$

$R_1 \rhd S_1$ We apply Proposition 60:

We have to show that

$$ep(R_1, n(\mathrm{X}) \wedge \neg(primes(\mathrm{P}) \wedge 2 \leqslant \mathrm{X} \leqslant \mathrm{P})) \ \rightarrow \ n(\mathrm{X})$$

which is true, since

$$ep(R_1, n(\mathrm{X}) \wedge \neg(primes(\mathrm{P}) \wedge 2 \leqslant \mathrm{X} \leqslant \mathrm{P})) \equiv$$
$$n(\mathrm{X}) + primes(\mathrm{P}) + max(\mathrm{M}) \wedge \mathrm{M} < \mathrm{P}$$

$R_1 \rhd S_2$ We apply Proposition 60:

We have to show that

$$ep(R_1, done \wedge [\neg primes(\mathrm{P}) \vee \neg max(\mathrm{P}) \vee (n(\mathrm{X}) \wedge 2 \leqslant \mathrm{Y} < \mathrm{X} \wedge \mathrm{Y} \mid \mathrm{X}]) \rightarrow done$$

which is true, by Proposition 39, since $ep(R_1, done) \rightarrow done$

$R_1 \rhd S_5$ We exploit the fact that R_1 does not consume $n(\_)$, and we prove the stronger $R_1 \rhd n(\mathrm{X})$ UNLESS $false$, obtaining a shorter proof. To apply Proposition 60, under the stronger assumption, we have to prove that

$$n(\mathrm{X}) \ \rightarrow \ (wp(R_1, n(\mathrm{X})) \vee \neg wp(R_1, true))$$

Indeed, $wp(R_1, n(\mathrm{X})) \vee \neg wp(R_1, true)$

$$= primes(\mathrm{P}) + max(\mathrm{M}) \ \wedge \ \mathrm{M} < \mathrm{P} \ \wedge \ [(n(\mathrm{X}) \ \wedge \ \mathrm{M} \neq \mathrm{X}-1) \ \vee \ \mathrm{M} = \mathrm{X}-1 \,]$$
$$\vee \ \neg(primes(\mathrm{P}) + max(\mathrm{M}) \ \wedge \ \mathrm{M} < \mathrm{P})$$

and this follows from $n(\mathrm{X})$, by simple propositional calculations.

6.3. *Proof of the eighth refinement step:* $E_7 \lhd E_8$

We show the most interesting proof, i.e. that S_2 is refined.

$R_3 \rhd_{H_3} S_2$ We apply Proposition 63:

We have to show that

$$ep(R_3, done + \mathscr{C}) \rightarrow done \vee \neg H_3$$

where $H_3 = \neg wp(R_1, true) \wedge \neg wp(R_2, true)$, according to Proposition 64. and $\mathscr{C} = \neg primes(\mathrm{P}) \vee \neg max(\mathrm{P}) \vee (n(\mathrm{X}) \wedge 2 \leqslant \mathrm{Y} < \mathrm{X} \wedge \mathrm{Y}|\mathrm{X})$

Since

$$ep(R_3, done + \mathscr{C}) = \neg done + \mathscr{C}$$

we have only to show that $\neg done + \mathscr{C} \rightarrow \neg H_3$. Then:

$$\begin{aligned}
\neg H_3 \equiv\ & wp(R_1, true) \vee wp(R_2, true) \\
\equiv\ & wp(R_1, true) \vee (n(\mathrm{X}) + n(\mathrm{Y}) \wedge \mathrm{Y}|\mathrm{X} \wedge \mathrm{Y} < \mathrm{X}) \\
\equiv\ & (S_5) \\
\equiv\ & wp(R_1, true) \vee (n(\mathrm{X}) \wedge 2 \leqslant \mathrm{Y} < \mathrm{X} \wedge \mathrm{Y}|\mathrm{X})] \\
\equiv\ & \\
& (primes(\mathrm{P}) + max(\mathrm{M}) \wedge \mathrm{M} < \mathrm{P}) \vee (n(\mathrm{X}) \wedge 2 \leqslant \mathrm{Y} < \mathrm{X} \wedge \mathrm{Y}|\mathrm{X})] \\
\Leftarrow\ & (S_3, S_9, s(L_{1.1})) \\
& \neg done + [\neg primes(\mathrm{P}) \vee \neg max(\mathrm{P}) \vee (n(\mathrm{X}) \wedge 2 \leqslant \mathrm{Y} < \mathrm{X} \wedge \mathrm{Y}|\mathrm{X})] \\
\equiv\ & \\
& \neg done + \mathscr{C}
\end{aligned}$$

7. Conclusions

Tuple space languages are acknowledged as a model to describe cooperation and, more generally, to program large distributed reactive systems.

To build a refinement calculus for this framework, we have defined: a variation of temporal logic to provide readable and concise specifications; an axiomatic semantics to derive properties of a tuple space system; an operational semantics to describe the allowed computations. The calculus is then based on the relation between these descriptions.

One of the contributions of our work is the idea to derive weakest preconditions by exploiting Back's *demonic strict* choice in non-deterministic selection. The transition system defining the operational semantics is based on the new notion of enabling precondition, which exploits the *angelic strict* choice. We use weakest and enabling preconditions to define the refinement of safety and liveness properties, respectively.

The calculus has been exploited to model and implement software processes, to describe software architectures, and may be extended to cope with software maintenance.

In [44] we provide evidence that the advantages of a refinement calculus are gained also for those programs known as *enactable process models*. The evidence is put forward by means of an example, a small Concurrent Engineering problem inspired by the ISPW-7 problem. In [45], we illustrate some refinement patterns within the tuple space framework. Since each pattern introduces an architectural style, they are called *architectural refinements*. We exemplify their use in the refinement of a particular system: a fragment of a software process. Then, the refinement calculus presented here provides a firm formal basis for a method that besides correctness, highlights the architectural choices in the design.

Acknowledgements

The referees of this paper and of the first author's Phd thesis (Krzysztof Apt, Dino Pedreschi, Gruia-Catalin Roman, and Franco Turini), provided many pertinent and useful suggestions. We gratefully thank all of them.

Appendix A. Proofs from Section 4

Proposition 28. *Let p and q be state formulae and R a rule. Then*

$$wp(R,\ p \wedge q) \equiv wp(R,\ p) \wedge wp(R,\ q)$$

Proof. We apply Definition 25 and show

(1) $wp(\text{TELL},\ p \wedge q) \equiv wp(\text{TELL},\ p) \wedge wp(\text{TELL},\ q)$
(2) $wp(\text{BODY},\ \{p_1 \wedge q_1,\ p_2 \wedge q_2\}) \equiv wp(\text{BODY},\ \{p_1,\ p_2\}) \wedge wp(\text{BODY},\ \{q_1,\ q_2\})$
(3) $wp(\text{CONS},\ p \wedge q) \equiv wp(\text{CONS},\ p) \wedge wp(\text{CONS},\ q)$
(4) $wp(\text{GUARD},\ p \wedge q) \equiv wp(\text{GUARD},\ p) \wedge wp(\text{GUARD},\ p)$

(1) For every **tell** $t(\bar{\text{B}})$ in TELL:

$$\begin{aligned}
&wp(\textbf{tell}\ t(\bar{\text{B}}),\ p \wedge q) \ \equiv \\
&ut(p \wedge q, t(\bar{\text{B}}))\,[t(\bar{\text{B}})^n / t(\bar{\text{B}})^{n-1}] \ \equiv \qquad \text{(Definition 21)} \\
&(ut(p, t(\bar{\text{B}})) \wedge ut(q, t(\bar{\text{B}})))\,[t(\bar{\text{B}})^n / t(\bar{\text{B}})^{n-1}] \ \equiv \\
&ut(p, t(\bar{\text{B}}))\,[t(\bar{\text{B}})^n / t(\bar{\text{B}})^{n-1}] \wedge ut(q, t(\bar{\text{B}}))\,[t(\bar{\text{B}})^n / t(\bar{\text{B}})^{n-1}] \ \equiv \\
&wp(\textbf{tell}\ t(\bar{\text{B}}),\ p) \wedge wp(\textbf{tell}\ t(\bar{\text{B}}),\ q)
\end{aligned}$$

(2) $wp(\textbf{body}\ t(\bar{\text{B}}),\ \{p_1 \wedge q_1,\ p_2 \wedge q_2\}) \ \equiv$

$$\begin{aligned}
&[t(\bar{\text{B}}) \wedge p_1 \wedge q_1] \vee [\neg t(\bar{\text{B}}) \wedge p_2 \wedge q_2)] \ \equiv \\
&\{[t(\bar{\text{B}}) \wedge p_1] \vee [\neg t(\bar{\text{B}}) \wedge p_2]\} \wedge \{[t(\bar{\text{B}}) \wedge q_1] \vee [\neg t(\bar{\text{B}}) \wedge q_2]\} \equiv \\
&wp(\textbf{body}\ t(\bar{\text{B}}),\ \{p_1,\ p_2\}) \wedge wp(\textbf{body}\ t(\bar{\text{B}}),\ \{q_1, q_2\})
\end{aligned}$$

(3) For every **cons** $t(\bar{B})$ in CONS, the next equivalence is shown in a way analogous to (1).

$$wp(\textbf{cons}\ t(\bar{B}),\ p \wedge q) \equiv wp(\textbf{cons}\ t(\bar{B}),\ p) \wedge wp(\textbf{cons}\ t(\bar{B}),\ q)$$

(4) $wp(\text{GUARD},\ p \wedge q) \equiv$

$$g2f(\text{GUARD}) \wedge (g2f(\text{GUARD}) \rightarrow p \wedge q) \equiv$$
$$g2f(\text{GUARD}) \wedge [g2f(\text{GUARD}) \rightarrow p] \wedge [g2f(\text{GUARD}) \rightarrow q] \equiv$$
$$wp(\text{GUARD}, p) \wedge wp(\text{GUARD}, P)$$

Proposition 29. *Let R be a rule, then $wp(R, false) \equiv false$.*

Proof. By straight application of Definition 25,
(1) For every **tell** $t(\bar{B})$ in TELL: $wp(\textbf{tell}\ t(\bar{B}), false) \equiv false$
(2) $wp(\text{BODY}, \{false, false\}) = false$
(3) For every **cons** $t(\bar{B})$ in CONSUME: $wp(\textbf{cons}\ t(\bar{B}), false) \equiv false$
(4) $wp(\text{GUARD}, false) \equiv g2f(\text{GUARD}) \wedge [g2f(\text{GUARD}) \rightarrow false] \equiv false$

Corollary 30. *Let p and q be state formulae and R a rule, then*

$$\frac{p \rightarrow q}{wp(R, p) \rightarrow wp(R, q)}$$

Proof.

$p \rightarrow q$
$\Rightarrow$ (Definition 14)
$p \equiv p \wedge q$
$\Rightarrow$
$wp(R, p) \equiv wp(R, p \wedge q)$
$\Rightarrow$ (Proposition 28)
$wp(R, p) \equiv wp(R, p) \wedge wp(R, q)$
$\Rightarrow$
$wp(R, p) \rightarrow wp(R, q)$ □

Corollary 31. *Let p be a state formula and R a rule, then*

$$wp(R, \neg p) \rightarrow \neg wp(R, p)$$

Proof. Assume $wp(R, \neg p) \wedge wp(R, p)$ and apply Propositions 28 and 29. □

Corollary 32. *Let p and q be state formulae and R a rule, then*

$$wp(R,\ p) \vee wp(R,\ q) \rightarrow wp(R,\ p \vee q)$$

Proof.

$$p \rightarrow p \vee q \quad \text{and} \quad q \rightarrow p \vee q$$
$$\Rightarrow \qquad \text{(Proposition 28)}$$
$$wp(R,\ p) \rightarrow wp(R,\ p \vee q) \quad \text{and} \quad wp(R,\ q) \rightarrow wp(R,\ p \vee q)$$
$$\Rightarrow$$
$$wp(R,\ p) \vee wp(R,\ q) \rightarrow wp(R,\ p \vee q) \qquad \square$$

Proposition 39. *Let p and q be state formulae and R a rule, then*

$$ep(R,\ p \vee q) \equiv ep(R,\ p) \vee ep(R,\ q)$$

Proof. The proof structure is similar to the one of Proposition 28
(1) For every **tell** $t(\bar{B})$ in TELL (similar to TELL in Proposition 28):

$$wp(\textbf{tell}\ t(\bar{B}),\ p \vee q) \equiv wp(\textbf{tell}\ t(\bar{B}\ p) \vee wp(\textbf{tell}\ t(\bar{B}\ q)$$

(2) $wp(\textbf{body}\ t(\bar{B}),\ \{p_1 \vee q_1,\ p_2 \vee q_2\}) \equiv$

$$[t(\bar{B}) \wedge (p_1 \vee q_1)] \vee [\neg t(\bar{B}) \wedge (p_2 \vee q_2)] \equiv$$
$$[t(\bar{B}) \wedge p_1] \vee [t(\bar{B}) \wedge q_1] \vee [\neg t(\bar{B}) \wedge p_2] \vee [\neg t(\bar{B}) \wedge q_2] \equiv$$
$$[t(\bar{B}) \wedge p_1] \vee [\neg t(\bar{B}) \wedge p_2] \vee [t(\bar{B}) \wedge q_1] \vee [\neg t(\bar{B}) \wedge q_2] \equiv$$
$$wp(\textbf{body}\ t(\bar{B}),\ \{p_1,\ p_2\}) \vee wp(\textbf{body}\ t(\bar{B}),\ \{q_1,\ q_2\})$$

(3) For every **cons** $t(\bar{B})$ we have

$$wp(\textbf{cons}\ t(\bar{B}),\ p \vee q) \equiv wp(\textbf{cons}\ t(\bar{B}),\ p) \vee wp(\textbf{cons}\ t(\bar{B}),\ q)$$

(4) $ep(\text{GUARD},\ p \vee q) \equiv$

$$g2f(\text{GUARD}) \wedge (p \vee q) \equiv$$
$$[g2f(\text{GUARD}) \wedge p] \vee [g2f(\text{GUARD}) \wedge q] \equiv$$
$$ep(\text{GUARD}, p) \vee ep(\text{GUARD}, p) \qquad \square$$

Corollary 40. *Let p and q be state formulae and R a rule, then*

$$\neg ep(R,\ p) \wedge ep(R,\ q) \rightarrow ep(R,\ q \wedge \neg p)$$

Proof. By contradiction.

$$\neg(\neg ep(R, p) \wedge ep(R, q) \rightarrow ep(R, q \wedge \neg p))$$
$$\equiv$$
$$\neg ep(R, p) \wedge \neg ep(R, q \wedge \neg p) \wedge ep(R, q)$$
$$\equiv$$
$$\neg ep(R, p) \wedge \neg ep(R, q \wedge \neg p) \wedge ep(R, q \wedge [p \vee \neg p])$$
$$\equiv$$
$$\neg ep(R, p) \wedge \neg ep(R, q \wedge \neg p) \wedge [ep(R, q \wedge p) \vee ep(R, q \wedge \neg p)]$$
$$\equiv$$
$$false \qquad \square$$

Corollary 41. *Let p and q be state formulae and R a rule, then*

$$\frac{p \rightarrow q}{ep(R, p) \rightarrow ep(R, q)}$$

Proof.

$$p \rightarrow q \overset{\text{Def. 14}}{\Rightarrow} q \equiv p \vee q \tag{A.1}$$

$$ep(R, p) \overset{\text{Prop. 39}}{\rightarrow} ep(R, p \vee q) \overset{\text{(A.1)}}{\equiv} ep(R, q) \qquad \square$$

Proposition 44. *Let p be a state formula and R a rule, then*

$$wp(R, q) \rightarrow ep(R, q)$$

Proof.

$$wp(\text{GUARD}, q) \equiv$$
$$g2f(\text{GUARD}) \wedge (g2f(\text{GUARD}) \rightarrow q) \rightarrow$$
$$g2f(\text{GUARD}) \wedge q \equiv$$
$$ep(\text{GUARD}, q) \qquad \square$$

Lemma 45. *Let R be a rule, then $wp(R, true) \equiv ep(R, true)$*

Proof. Let G denote $g2f(\text{GUARD})$:

$$wp(R, true) \equiv \qquad \text{(Definition 25)}$$
$$G \wedge [G \rightarrow true] \equiv$$
$$G \equiv$$

$$\begin{aligned} &\text{G} \wedge \mathit{true} \equiv && \text{(Definition 33)} \\ &\mathit{ep}(R, \mathit{true}) && \square \end{aligned}$$

Proposition 46. *Let R be a rule and q a state formula, then*

$$\mathit{wp}(R, q) \equiv \neg \mathit{ep}(R, \neg q) \wedge \mathit{ep}(R, \mathit{true})$$

Proof. The proof structure is similar to the one of Proposition 28.

(1) For every **tell** $t(\bar{\text{B}})$ in TELL:

$$\begin{aligned} &\mathit{wp}(\ \textbf{tell}\ t(\bar{\text{B}}),\ \neg q) \equiv \mathit{ut}(\neg q, t(\bar{\text{B}}))\,[t(\bar{\text{B}})^n / t(\bar{\text{B}})^{n-1}] \equiv \\ &\neg \mathit{ut}(q, t(\bar{\text{B}}))\,[t(\bar{\text{B}})^n / t(\bar{\text{B}})^{n-1}] \equiv \neg \mathit{wp}(\textbf{tell}\ t(\bar{\text{B}}),\ q) \end{aligned}$$

(2) $\mathit{wp}(\textbf{body}\ t(\bar{\text{B}}),\ \{\neg q_1, \neg q_2\}) \equiv$

$$\begin{aligned} &[t(\bar{\text{B}}) \wedge \neg q_1] \vee [\neg t(\bar{\text{B}}) \wedge \neg q_2)] \equiv \\ &[\neg t(\bar{\text{B}}) \vee \neg q_1] \wedge [t(\bar{\text{B}}) \vee \neg q_2]\} \equiv \\ &\neg [t(\bar{\text{B}}) \wedge q_1] \wedge \neg [\neg t(\bar{\text{B}}) \wedge q_2]\} \equiv \\ &\neg\, \mathit{wp}(\textbf{body}\ t(\bar{\text{B}}),\ \{q_1, q_2\}) \end{aligned}$$

(3) For every **cons** $t(\bar{\text{B}})$ in TELL, $\mathit{wp}(\textbf{cons}\ t(\bar{\text{B}}),\ \neg q) \equiv \neg\, \mathit{wp}(\textbf{cons}\ t(\bar{\text{B}}),\ q)$ is shown in a way analogous to (1)

(4) $\mathit{ep}(\text{GUARD}, \mathit{true}) \wedge \neg\, \mathit{ep}(\text{GUARD}, \neg q) \equiv$

$$\begin{aligned} &g2f(\text{GUARD}) \wedge \neg [g2f(\text{GUARD}) \wedge \neg q) \equiv \\ &g2f(\text{GUARD}) \wedge [g2f(\text{GUARD}) \rightarrow q] \equiv \\ &\mathit{wp}(\text{GUARD}, q) \qquad \square \end{aligned}$$

Proposition 47. *Let s be a state and R a rule, then*

$$s \models \mathit{wp}(R, \mathit{true}) \Leftrightarrow T(R)(s) \neq \emptyset$$

Proof.

"$\Leftarrow$":

$$\begin{aligned} &T(R)(s) \neq \emptyset \\ &\quad \Rightarrow && \text{(Let } s' \in T(R)(s), \text{ surely } s' \models \mathit{true}. \text{ Then, Definition 42)} \\ &s \models \mathit{ep}(R, \mathit{true}) \\ &\quad \Leftrightarrow && \text{(Lemma 45)} \\ &s \models \mathit{wp}(R, \mathit{true}) \end{aligned}$$

"⇒": Take a Universe of interpretations $\mathcal{U} = \{p_1, \ldots, p_i, \ldots\}$, where the p_i's are ground atoms.

Since, by hypothesis $s| \models ep(R, true)$, if we take p_1, then an n_1 exists such that $s| \models ep(R, p_1^{n_1})$ and $s| \not\models ep(R, p_1^{n_1+1})$.

By Corollary 40, $s| \models ep(R, p_1^{n_1} \wedge \neg p_1^{n_1+1})$.

We then consider p_2. An n_2 exists such that $s| \models ep(R, p_1^{n_1} \wedge \neg p_1^{n_1+1} \wedge p_2^{n_2})$ and $s| \not\models ep(R, p_2^{n_2+1})$. We apply again Corollary 40, and continue this construction till finding the formula $r = \bigwedge_{p_i \in \mathcal{U}} p_i^{n_i} \wedge \neg p_i^{n_i+1}$. By construction $s| \models ep(R, r)$.

We take $s' = \{n_i$ occurrences of p_i, for all $i\}$. Then, for all q:

$s'| \models q$

$\Rightarrow$ (The only model for r is s')

$r \rightarrow q$

$\Rightarrow$ ($s| \models ep(R, r)$, Corollary 41)

$s| \models ep(R, q)$

Hence, by Definition 42, $s' \in T(R)(s)$. □

Theorems 50 and 49. Let R be a rule, s a state, and q a state formula, then $\forall \bar{Y}$ free in q:

$$s| \models wp(R, q(\bar{Y})) \Leftrightarrow T(R)(s) \neq \emptyset \wedge \forall s'.[s' \in T(R)(s) \rightarrow s'| \models q(\bar{Y})]$$
$$s| \models ep(R, q(\bar{Y})) \Leftrightarrow \exists s'.[s' \in T(R)(s) \wedge s'| \models q(\bar{Y})]$$

Proof. *Part a*: $s| \models wp(R, q(\bar{Y})) \Rightarrow T(R)(s) \neq \emptyset \wedge \forall s'.[s' \in T(R)(s) \rightarrow s'| \models q(\bar{Y})]$

Proposition 47 says that

$$s| \models wp(R, q(\bar{Z})) \quad \Rightarrow \quad T(R)(s) \neq \emptyset$$

We now show that for all $\bar{Z}$ free in q:

$$s| \models wp(R, q(\bar{Z})) \quad \Rightarrow \quad \forall s' \in T(R)(s).s'| \models q(\bar{Z})$$

Let us call G $g2f$(GUARD), B the BODY, T the TELL, and C the CONSUME of rule R:

$s'| \models \neg q(\bar{Z}) \wedge s' \in T(R)(s)$

$\Rightarrow$ (Definition 42)

$s| \models ep(R, \neg q(\bar{Z}))$

$\Leftrightarrow$ (Definition 33)

$s| \models (G \wedge wp(C, wp(B, \{wp(T_1, \neg q(\bar{Z})), wp(T_2, \neg q(\bar{Z}))\})))$

$\Leftrightarrow$ (see the proof of Proposition 46, equivalence 1)

$s| \models G \wedge wp(C, wp(B, \{\neg wp(T_1, q(\bar{Z})), \neg wp(T_2, q(\bar{Z}))\}))$

$$\Leftrightarrow \qquad \text{(Definition 25, let } q_i \equiv wp(\mathrm{T}_i, q(\bar{z}))\text{)}$$

$$s \models \mathrm{G} \wedge wp(\mathrm{C}, wp(\mathrm{B}, \{\neg q_1, \neg q_2\}))$$

$$\Leftrightarrow \qquad \text{(Definition 25)}$$

$$s \models \mathrm{G} \wedge wp(\mathrm{C}, [\mathrm{B} \wedge \neg q_1] \vee [\neg \mathrm{B} \wedge \neg q_2])$$

$$\Leftrightarrow \qquad \text{(See the proof of Proposition 28)}$$

$$s \models \mathrm{G} \wedge [(wp(\mathrm{C}, \mathrm{B}) \wedge \neg wp(\mathrm{C}, q_1)) \vee (\neg wp(\mathrm{C}, \mathrm{B}) \wedge \neg wp(\mathrm{C}, q_2))] \tag{A.2}$$

On the other side,

$$s \models wp(R, q(\bar{z}))$$

$$\Leftrightarrow \qquad \text{(Definition 25, } q_i \text{ see above)}$$

$$s \models \mathrm{G} \wedge [\mathrm{G} \rightarrow wp(\mathrm{C}, wp(\mathrm{B}, \{q_1, q_2\}))]$$

$$\Leftrightarrow \qquad \text{(Definition 25)}$$

$$s \models \mathrm{G} \wedge [\mathrm{G} \rightarrow wp(\mathrm{C}, [\mathrm{B} \wedge q_1] \vee [\neg \mathrm{B} \wedge q_2])]$$

$$\Rightarrow \qquad \text{(See the proof of Proposition 28)}$$

$$s \models \neg wp(\mathrm{C}, \mathrm{B}) \wedge wp(\mathrm{C}, q_1) \wedge [wp(\mathrm{C}, \mathrm{B}) \vee wp(\mathrm{C}, q_2)]$$

Hence the contradiction with (A.2).

Part b: $s \models ep(R, q(\bar{\mathrm{Y}})) \Rightarrow \exists s'.[s' \in T(R)(s) \wedge s' \models q(\bar{\mathrm{Y}})]$

Let us consider the case of a rule without the body, the proof for the general case is an extension exploiting the same ideas. $\forall$ fixed $\bar{\mathrm{Y}}$ free in q:

$$s \models ep(R, q) \overset{\text{def } 33}{\equiv} s \models \mathrm{G} \wedge C(q)$$

where $C(q) = wp(\text{CONSUME}, wp(\text{TELL}, q))$, i.e. a syntactic transformation of q, $\mathrm{G} = g2f(\text{GUARD})$.

By language definition, all the variables appearing in the TELL of a rule also appear positively in the rule GUARD. Let $\bar{\mathrm{X}}$ be these variables, by Definition 25, $\bar{\mathrm{X}}$ covers all the variables free in $C(q)$.

$$s \models \mathrm{G} \wedge C(q)$$

$$\equiv \qquad (\vartheta \text{ grounding for } \bar{\mathrm{X}})$$

$$s \models \exists \vartheta.\ (\mathrm{G}\vartheta \wedge C(q)\vartheta)$$

$$\equiv \qquad \text{(Definition 14)}$$

$$\exists \vartheta.\ s \models (\mathrm{G}\vartheta \wedge C(q)\vartheta)$$

Let us consider now the rule instance $R\vartheta$. This has a deterministic behavior. Moreover, (see the comment to Eq. 3 in Definition 25), $q\vartheta = q$. Hence,

$$s \models \mathrm{G}\vartheta \wedge C(q)\vartheta \equiv s \models ep(R\vartheta, q) \equiv s \models wp(R\vartheta, q) \tag{A.3}$$

Now

$$s \Vdash wp(R\vartheta, q)$$
$$\Rightarrow \qquad \text{(See Part a)}$$
$$T(R\vartheta)(s) \neq \emptyset \wedge \forall s'.[s' \in T(R\vartheta)(s) \rightarrow s' \Vdash q]$$
$$\Rightarrow$$
$$\exists s'.\,[s' \in T(R\vartheta)(s) \wedge s' \Vdash q]$$

We conclude by establishing that $T(R\vartheta)(s) \subseteq T(R)(s)$. Let $\bar{s} \in T(R\vartheta)(s)$, by Definition 42,

$$\forall p.\forall \bar{z} \text{ free in } p.\ \bar{s} \Vdash p \Rightarrow s \Vdash ep(R\vartheta, p)$$

Now

$$s \Vdash ep(R\vartheta, p)$$
$$\Rightarrow \qquad \text{(see (A.3))}$$
$$s \Vdash \mathrm{G}\vartheta \wedge C(p)\vartheta$$
$$\Rightarrow$$
$$s \Vdash \exists\vartheta\,(\mathrm{G} \wedge C(p))\vartheta$$
$$\Rightarrow \qquad \text{(Definition 33)}$$
$$s \Vdash \exists\vartheta\, ep(R, p)\vartheta$$
$$\Rightarrow \qquad \text{(Definition 14)}$$
$$s \Vdash ep(R, p)$$

Hence $\forall p.\forall \bar{z}$ free in $p.\ \bar{s} \Vdash p \Rightarrow s \Vdash ep(R, p)$, and so $\bar{s} \in T(R)(s)$

Part c: $s \Vdash ep(R, q(\bar{\gamma})) \Leftarrow \exists s'.[s' \in T(R)(s) \wedge s' \Vdash q(\bar{\gamma})]$
Derives directly from Definition 42.

Part d: $s \Vdash wp(R, q(\bar{\gamma})) \Leftarrow T(R)(s) \neq \emptyset \wedge \forall s'.[s' \in T(R)(s) \rightarrow s' \Vdash q(\bar{\gamma})]$
By contraposition.

$$s \not\Vdash wp(R, q(\bar{\gamma}))$$
$$\Rightarrow \quad \text{(Proposition 46)}$$
$$s \Vdash ep(R, \neg q(\bar{\gamma})) \ \vee \ s \not\Vdash ep(R, true)$$
$$\Rightarrow \quad (T(R)(s) \neq \emptyset \overset{\text{Prop. 47}}{\Rightarrow} s \Vdash wp(R, true) \overset{\text{Lemma 45}}{\Rightarrow} s \Vdash ep(R, true))$$
$$s \Vdash ep(R, \neg q(\bar{\gamma}))$$

$\Rightarrow$ (see *Part b*)

$\exists s'.[s' \in T(R)(s) \wedge s' \not\models \neg q(\bar{Y})]$ □

References

[1] M. Abadi, L. Lamport, Composing specifications, in: J.W. de Bakker, W.P. de Roever, G. Rozenberg, (Eds.), Proc. Stepwise Refinement of Distributed Systems, Lecture Notes in Computer Science, Vol. 430, Springer, Berlin, 1989, pp. 1–41.

[2] B. Alpern, F.B. Shneider, Defining liveness, Inform. Process Lett. 21 (4) (1985) 181–185.

[3] V. Ambriola, P. Ciancarini, C. Montangero, Software process enactment in Oikos, in: R. Taylor (Ed.), Proc. ACM SIGSOFT 90, 4th Symp. on Software Development Environments, Irvine, CA, 1990, pp. 183–192.

[4] V. Ambriola, G.A. Cignoni, L. Semini, Eta – everything buT assignment, in: M. Alpuente, R. Barbuti, I. Ramos (Eds.), Proc. Joint Conf. on Declarative Programming GULP-PRODE'94, Peniscula, September 1994.

[5] V. Ambriola, G.A. Cignoni, L. Semini, A proposal to merge object orientation, logic programming, and multiple tuple spaces, Comput. Languages 22 (2/3) (1996) 79–93.

[6] V. Ambriola, L. Semini, control specification in tuple space based languages, Technical Report 19-93, Dipartimento di Informatica, Università di Pisa, 1993.

[7] J.M. Andreoli, R. Pareschi, Linear objects: logical processes with built-in inheritance, in: D.H.D. Warren, P. Szeredi (Eds.), Proc. 7th Internat. Conf. on Logic Programming, MIT Press, Cambridge, MA, 1990, pp. 495–510.

[8] K.R. Apt, E.-R. Olderog, Verification of Sequential and Concurrent Programs, Springer, Berlin, 1991.

[9] F. Arbab, I. Herman, P. Spilling, An overview of manifold and its implementation, Concurrency: Practice Experience 5 (1) (1993) 23–70.

[10] R.J.R. Back, R. Kurki-Suonio, Distributed cooperation with action systems, ACM Trans. Programm. Languages Systems 10 (4) (1988) 513–554.

[11] R.J.R. Back, J. von Wright, Refinement calculus, Part I: sequential nondeterministic programs, in: J.W. de Bakker, W.P. de Roever, G. Rozenberg (Eds.), Proc. Stepwise Refinement of Distributed Systems, Lecture Notes in Computer Science, Vol. 430, Springer, Berlin, 1989, pp. 42–66.

[12] J.-P. Banâtre, D. Le Métayer, The GAMMA model and its discipline of programming, Science of Computer Programming 15 (1990) 55–77.

[13] J.P. Banatre, D. Le Métaier, Programming by multiset transformation, Commun. ACM 16 (1) (1993) 55–77.

[14] K. De Bosschere, J.-M. Jacquet, Multi-prolog: definition, operational semantics and implementation, in: D.S. Warren (Ed.), Proc. 10th Internat. Conf. on Logic Programming, 1993, pp. 299–313.

[15] K. De Bosschere, J.-M. Jacquet. μ^2 log: towards remote coordination, in: P. Ciancarini, C. Hankin (Eds.), Proc. Coordination Languages and Models, 1st Int. Conf., COORDINATION 96, Lecture Notes in Computer Science, Vol. 1061, Cesena, Springer, Berlin, April 1996, pp. 142–159.

[16] A. Brogi, P. Ciancarini, The concurrent language shared prolog, ACM Trans. Programm. Languages Systems 13 (1) (1991) 99–123.

[17] M. Broy, Functional specification of time sensitive communicating systems, in: J.W. de Bakker, W.P. de Roever, G. Rozenberg (Eds.), Proc. Stepwise Refinement of Distributed Systems, Lecture Notes in Computer Science, Vol. 430, Springer, Berlin, 1989, pp. 153–179.

[18] M. Broy, (Inter-) Action refinement: the easy way, in: M. Broy (Ed.), Program Design Calculi, NATO ASI Series, Springer, Berlin, 1993, pp. 121–158.

[19] A. Bucci, P. Ciancarini, C. Montangero, A distributed logic language based on multiple tuple spaces, Proc. Logic Programming Conf., Tokio, 1991, pp. 199–208.

[20] N. Carriero, D. Gelernter, Linda and friends, IEEE Comput. 19 (8) (1986) 26–34.

[21] N. Carriero, D. Gelernter, Coordination languages and their significance, Commun. ACM 5 (2) (1989) 97–107.

[22] N. Carriero, D. Gelernter, Linda in context, Commun. ACM 32 (4) (1989) 444–459.

[23] K.M. Chandy, J. Misra, Parallel Program Design: A Foundation, Addison-Wesley, Reading, MA, 1988.

[24] X.J. Chen, Formalism and verification towards software process modelling, Ph.D. Thesis, Scuola Normale Superiore di Pisa, 1992.

[25] X.J. Chen, C. Montangero, Compositional refinements of multiple blackboard systems, Acta Inform. 32 (5) (1995) 415–458.

[26] P. Ciancarini, PoliS: a programming model for multiple tuple spaces, in: Proc. 6th IEEE Internat. Workshop on Software Specification and Design, 1991.

[27] P. Ciancarini, C. Hankin (Eds.), Proc. COORDINATION 96, Lecture Notes in Computer Science, Vol. 1061, Cesena, Springer, Berlin, April 1996.

[28] H.C. Cunningam, G.-C. Roman, A UNITY-style programming logic for shared dataspace programs, IEEE Trans. Parallel Distributed Systems 1 (3) (1990) 365–376.

[29] E.W. Dijkstra, C.S. Scholten, Predicate Calculus and Program Semantics, Springer, Berlin, 1990.

[30] A. Finkelstein, J. Kramer, B. Nuseibeh (Eds.), Software Process Modelling and Technology, Research Study Press distributed by Wiley, London, 1994.

[31] D. Garlan, D. LeMetayer (Eds.), Proc. 2nd Internat. Conf. on Coordination Models and Languages, Lecture Notes in Computer Science, Vol. 1282, Berlin, Germany, Springer, Berlin, 1997.

[32] M. Gaspari, L. Semini, Fairness in logic languages based on shared dataspace, in: K. De Bosschere, J.-M. Jacquet, P. Tarau (Eds.), Proc. ICLP'93 Post-Conf. Workshop on Blackboard-Based Logic Programming, Budapest, 1993.

[33] J.-Y. Girard, Linear logic, Theore. Comput. Sci. 50 (1987) 1–102.

[34] H.J.M. Goeman, J.N. Kok, K. Sere, R. Udkin, Coordination in the ImpUNITY framework, Science of Computer Programming 31 (2–3) (1998) 313–334.

[35] C.A.R. Hoare, Programs are predicates, in: C.A.R. Hoare, J.C. Shepherdson (Eds.), Mathematical Logic and Programming Languages, Prentice-Hall, Englewood Cliffs, NJ, 1985, pp. 141–155.

[36] C.A.R. Hoare, Algebra and models, in: M. Broy (Ed.), Program Design Calculi, NATO ASI Series, Springer, Berlin, 1993, pp. 161–195.

[37] J.-M. Jacquet, K. De Bosschere, On composing concurrent logic processes, in: L. Sterling (Ed.), Proc. 12th Internat. Conf. on Logic Programming, ICLP'95, Tokio, MIT Press, Cambridge, MA, 1995, pp. 531–545.

[38] C.B. Jones, Specification and design of (parallel) programs, in: R.E.A. Mason (Ed.), Information Processing 83, North-Holland, Amsterdam, 1983, pp. 321–332.

[39] F. Kröger, Temporal Logic of Programs, Springer, Berlin, 1987.

[40] J.W. Lloyd, Foundations of Logic Programming, Springer, Berlin, 1987.

[41] Z. Manna, A. Pnueli, Verification of concurrent programs: a temporal proof system, in: J.W. de Bakker, J. van Leeuwen (Eds.), Foundations of Computer Science IV, Distributed Systems: Part 2, Mathematical Centre Tracts, Vol. 159, Amsterdam, 1983, pp. 163–255.

[42] Z. Manna, A. Pnueli, the Temporal Logic of Reactive and Concurrent Systems, Springer, New York, 1992.

[43] C. Montangero, V. Ambriola, Oikos: constructing process-centered SDEs, in: A. Finkelstein, J. Kramer, B. Nuseibeh (Eds.), Software Process Modelling and Technology, Research Study Press distributed by Wiley, London, 1994, pp. 131–151.

[44] C. Montangero, L. Semini, Applying refinement calculi to software process modelling, in: W. Schaefer (Ed.), Proc. 4th Internat. Conf. on the Software Process ICSP4, Brighton, UK, IEEE Computer Society Press, Los Alamitos, December 1996, pp. 63–74.

[45] C. Montangero, L. Semini, Refining by architectural styles or architecting by refinements, in: L. Vidal, A. Finkelstein, G. Spanoudakis, A.L. Wolf (Eds.), 2nd Internat. Software Architecture Workshop, Proc. SIGSOFT '96 Workshops, Part 1, San Francisco, CA, October 1996, ACM Press, New York, pp. 76–79.

[46] L. Monteiro, A. Porto, Entailment based actions for coordination, Theoret. Comput. Sci. 192 (2) (1998) 259–286.

[47] C.C. Morgan, Programming from Specifications, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[48] C.C. Morgan, The refinement calculus, in: M. Broy (Ed.), Program Design Calculi, NATO ASI Series, Springer, Berlin, 1993, pp. 3–52.

[49] A. Pnueli, Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. in: Current Trends in Concurrency, Lecture Notes in Computer Science, Vol. 224, Springer, Berlin, 1986, pp. 510–584.

[50] A. Porto, V. Vasconcelos, Truth and action osmosis (The TAO Computational Model), in: J.M. Andreoli, C. Hankin, D. LeMetayer (Eds.), Coordination Programming: Mechanisms, Models and Semantics, Imperial College Press, 1996, pp. 65–97.
[51] G.-C. Roman, H.C. Cunningam, Mixed programming metaphors in a shared dataspace model of concurrency, IEEE Trans. Software Eng. 16 (2) (1990) 1361–1376.
[52] G.-C. Roman, R.F. Gamble, W.E. Ball, Formal derivation of rule-based programs, IEEE Trans. Software Eng. 19 (3) (1993) 277–296.
[53] L. Semini, Refinement in tuple space languages. Ph.D. Thesis, Dipartimento di Informatica, Università di Pisa, 1996, TD 10-96.
[54] L. Sterling, E.Y. Shapiro, The Art of Prolog, 2nd ed., MIT Press, Cambridge, MA, 1994.